cfplot User Guide

Introduction

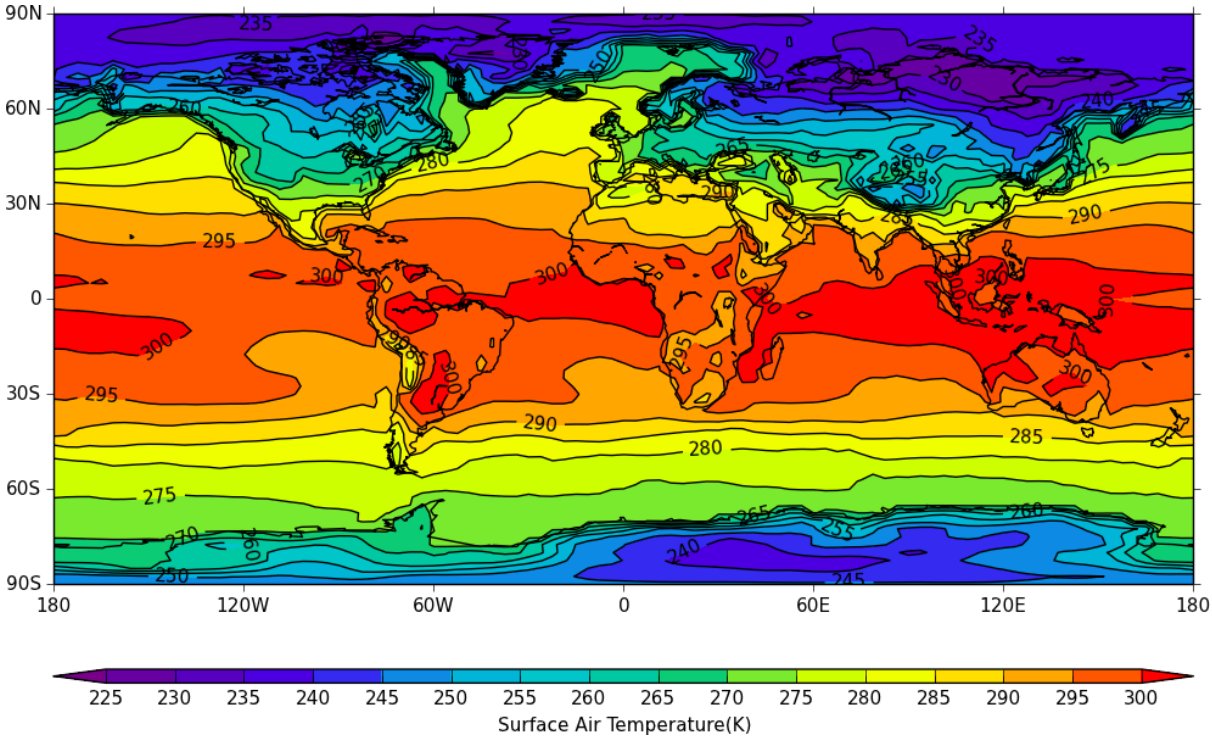
cfplot is a set of Python routines for making the common contour and vector plots that climate researchers use. cfplot generally uses cf-python to present the data and CF attributes for plotting as two-dimensional data fields.

To run the following examples in the Met department at the University of Reading setup the Python paths.
setup canopy

Outside of the Meteorology department you'll need to follow the cfplot installation instructions

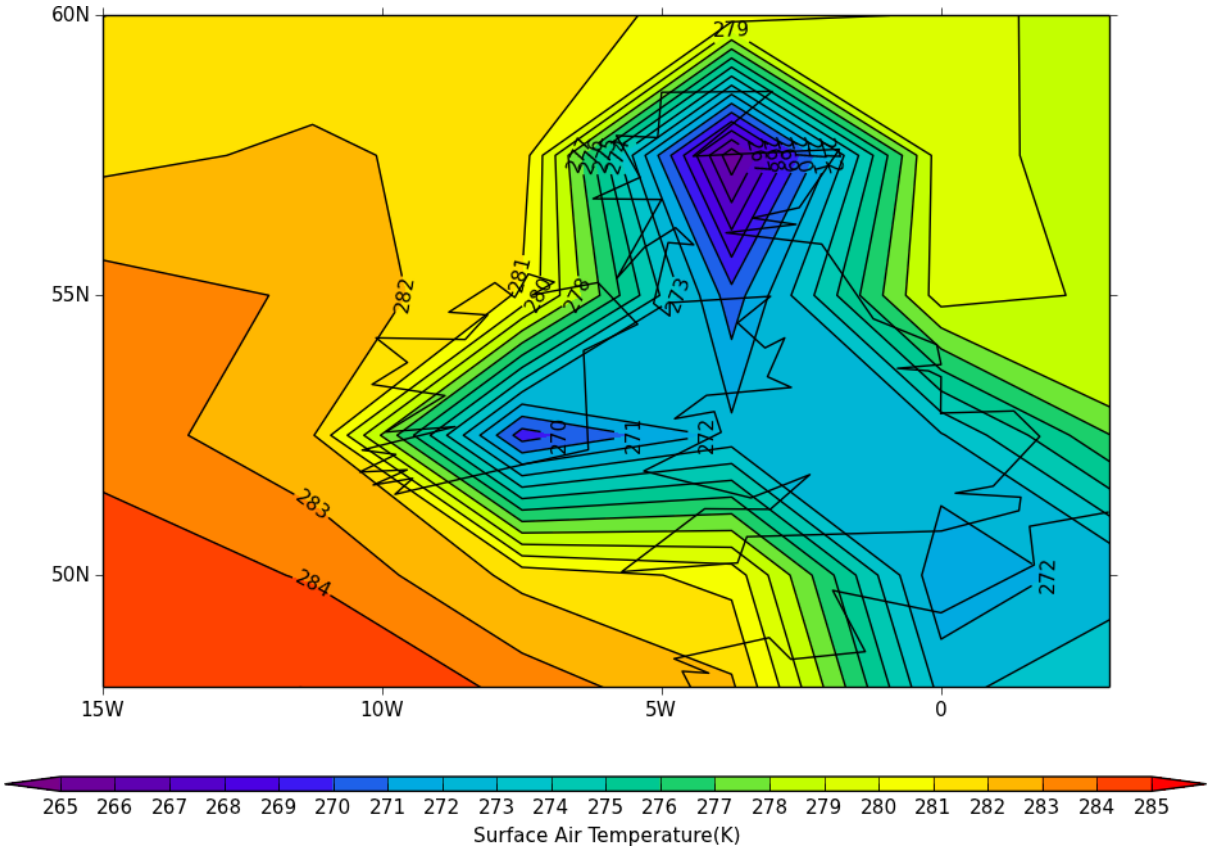
The data to make a contour plot can be read in and passed to cfplot using cf-python as per the following example.

```
import cf, cfplot as cfp
f=cf.read('/opt/graphics/cfplot_data/tas_A1.nc')[0]
cfp.con(f.subspace(time=15))
```



The **mapset** routine is used to change the map area and projection.
cfp.mapset(lonmin=-15, lonmax=3, latmin=48, latmax=60) sets the map to a view over the British Isles. Further plots will use the same map projection and limits. To reset the mapping to the default cylindrical projection -180 to 180 in longitude and -90 to 90 in latitude use **cfp.mapset()**.

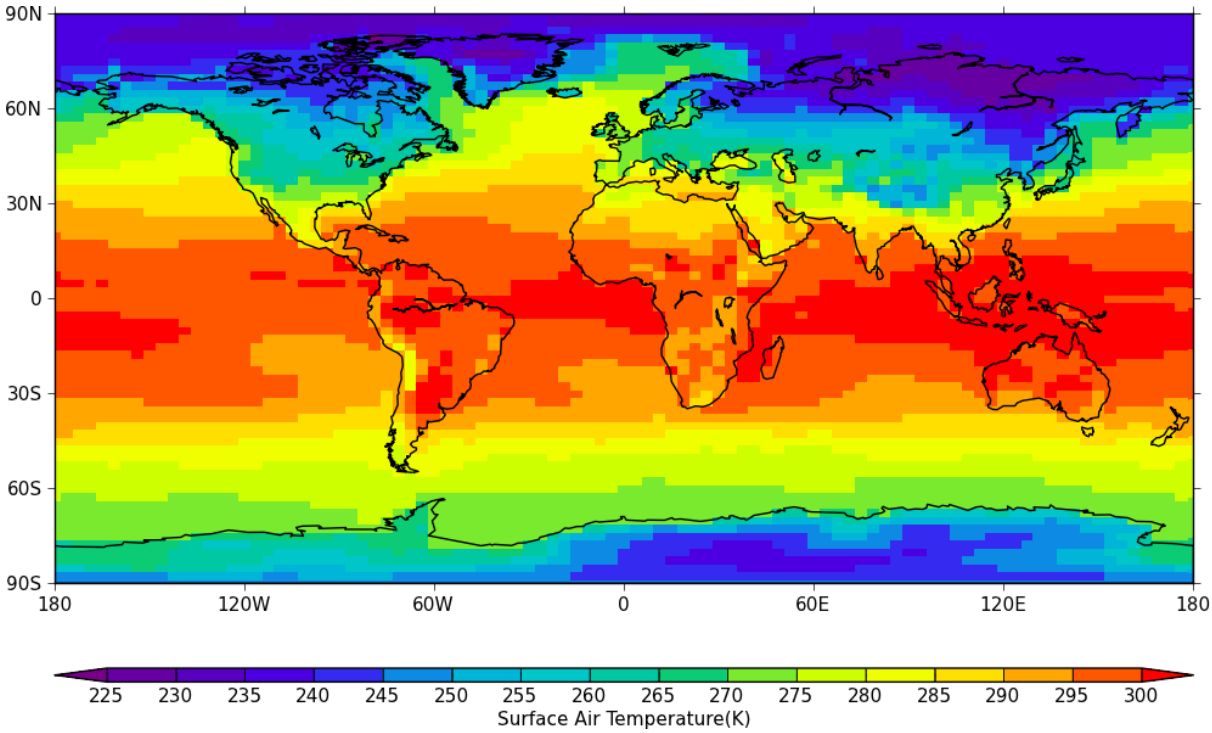
```
import cf, cfplot as cfp
f=cf.read('/opt/graphics/cfplot_data/tas_A1.nc')[0]
cfp.mapset(lonmin=-15, lonmax=3, latmin=48, latmax=60)
cfp.levs(min=265, max=285, step=1)
cfp.con(f.subspace(time=15))
```



The default settings are for colour fill and contour lines. These can be changed with the **fill=0** and **lines=0** flags to **con**.

Blockfill plots in the cylindrical projection are made using the **blockfill=1** flag to the **con** routine.

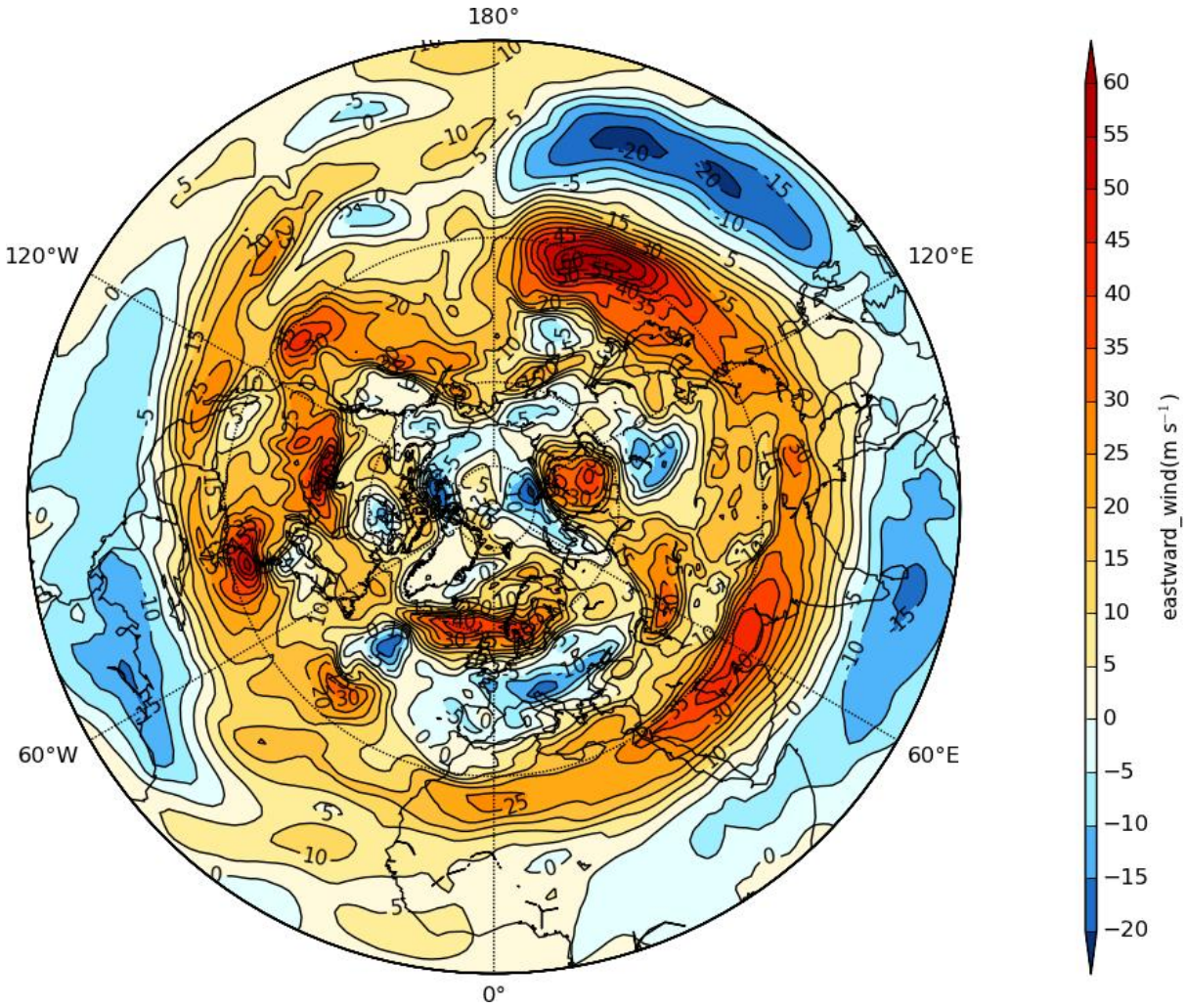
```
import cf, cfplot as cfp
f=cf.read('/opt/graphics/cfplot_data/tas_A1.nc')[0]
cfp.con(f.subspace(time=15), blockfill=1)
```



Blockfill in the polar stereographic projection isn't supported yet.

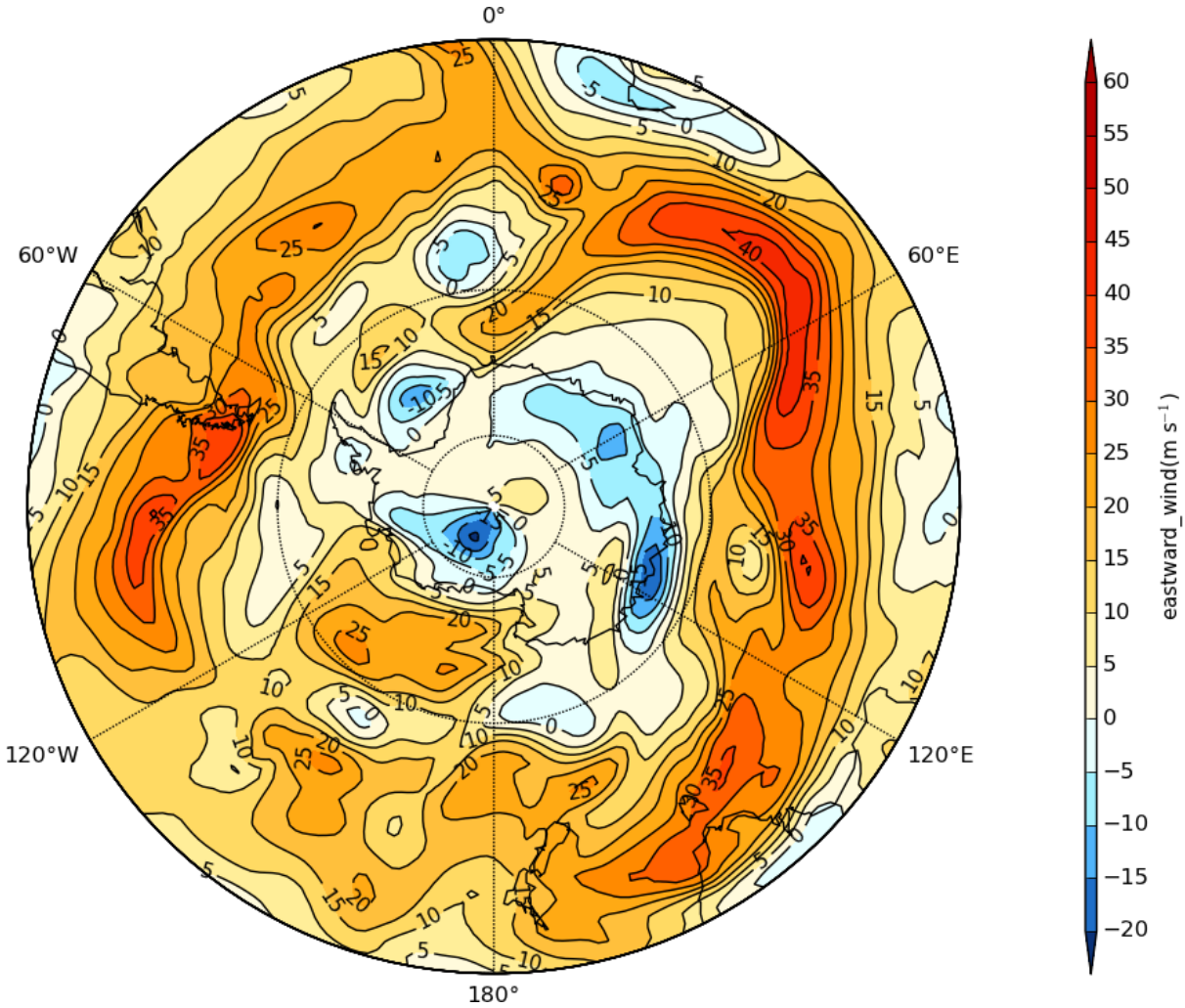
Polar Stereographic plots are set using **proj='npstere'** or **proj='spstere'** in the call to **mapset**.

```
import cf, cfplot as cfp
f=cf.read('/opt/graphics/cfplot_data/ggap.nc')[7]
cfp.mapset(proj='npstere')
cfp.con(f.subspace(pressure=500))
```



The **mapset bounding_lat** and **lon_0** parameters are used to set the latitude limit of the plot and the orientation of the plot. Generally for the southern hemisphere the Greenwich Meridian (zero degrees longitude) is plotted at the top of the plot and is set with lon_0=180.

import cf, cfplot as cfp
f=cf.read('/opt/graphics/cfplot_data/ggap.nc')[7]
cfp.mapset(proj='spstere', boundinglat=-30, lon_0=180)
cfp.con(f.subspace(pressure=500))



Passing data to cfplot

Using cf-python - CF compliant data

Data is generally passed to cfplot for ploting via cf-python. Contour and vector plots require a 2-dimensional field. cf-python is very flexible and can be used to select fields, levels, times, means for both CF and non-CF compliant data.

CF data is data that follows the NetCDF Climate and Forecast (CF) Metadata Conventions. The conventions define metadata that provide a definitive description of what the data in each variable represents, and of the spatial and temporal properties of the data.

import cf, cfplot as cfp
f=cf.read('/opt/graphics/cfplot_data/ggap.nc')

f is now an list of 12 fields.

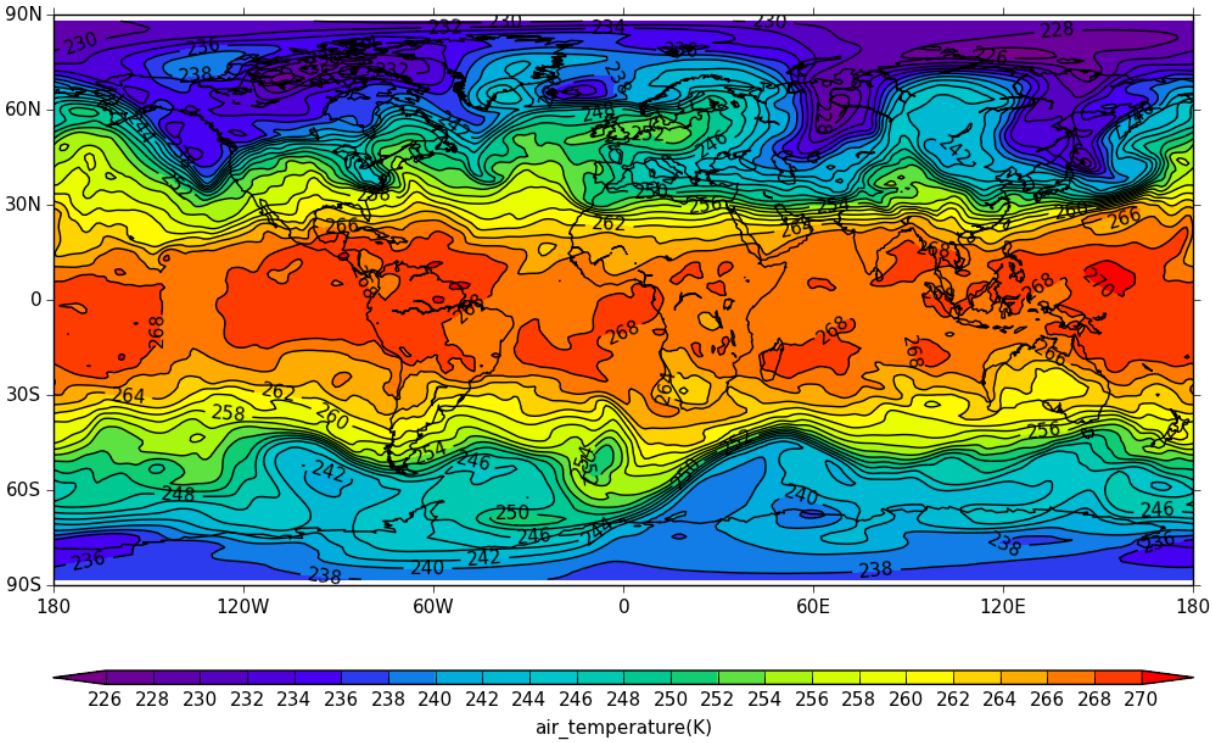
[<CF Field: long_name:Potential vorticity(time(1), pressure(23), latitude(160), longitude(320)) K m2 kg**-1 s**-1>,**
<CF Field: long_name:Ozone mass mixing ratio(time(1), pressure(23), latitude(160), longitude(320)) kg kg-1>,**
<CF Field: air_temperature(time(1), pressure(23), latitude(160), longitude(320)) K>,
<CF Field: atmosphere_relative_vorticity(time(1), pressure(23), latitude(160), longitude(320)) s-1>,**
<CF Field: atmosphere_relative_vorticity(time(1), pressure(23), latitude(160), longitude(320)) m2 s**-1>,**
<CF Field: divergence_of_wind(time(1), pressure(23), latitude(160), longitude(320)) s-1>,**
<CF Field: divergence_of_wind(time(1), pressure(23), latitude(160), longitude(320)) m2 s**-1>,**
<CF Field: eastward_wind(time(1), pressure(23), latitude(160), longitude(320)) m s-1>,**
<CF Field: geopotential(time(1), pressure(23), latitude(160), longitude(320)) m2 s**-2>,**
<CF Field: northward_wind(time(1), pressure(23), latitude(160), longitude(320)) m s-1>,**
<CF Field: relative_humidity(time(1), pressure(23), latitude(160), longitude(320)) %>,
<CF Field: specific_humidity(time(1), pressure(23), latitude(160), longitude(320)) kg kg-1>,**
<CF Field: vertical_air_velocity_expressed_as_tendency_of_pressure(time(1), pressure(23), latitude(160), longitude(320)) Pa s-1>]**

To see what levels are available in the data use f[2].item('pressure').array

array([1000., 925., 850., 775., 700., 600., 500., 400.,
300., 250., 200., 150., 100., 70., 50., 30.,
20., 10., 7., 5., 3., 2., 1.], dtype=float32)

In the case below we select the 500mb temperature with the cf subspace method.

f[2].subspace(pressure=500)
<CF Field: air_temperature(time(1), pressure(1), latitude(160), longitude(320)) K>
cfp.con(f[2].subspace(pressure=500))

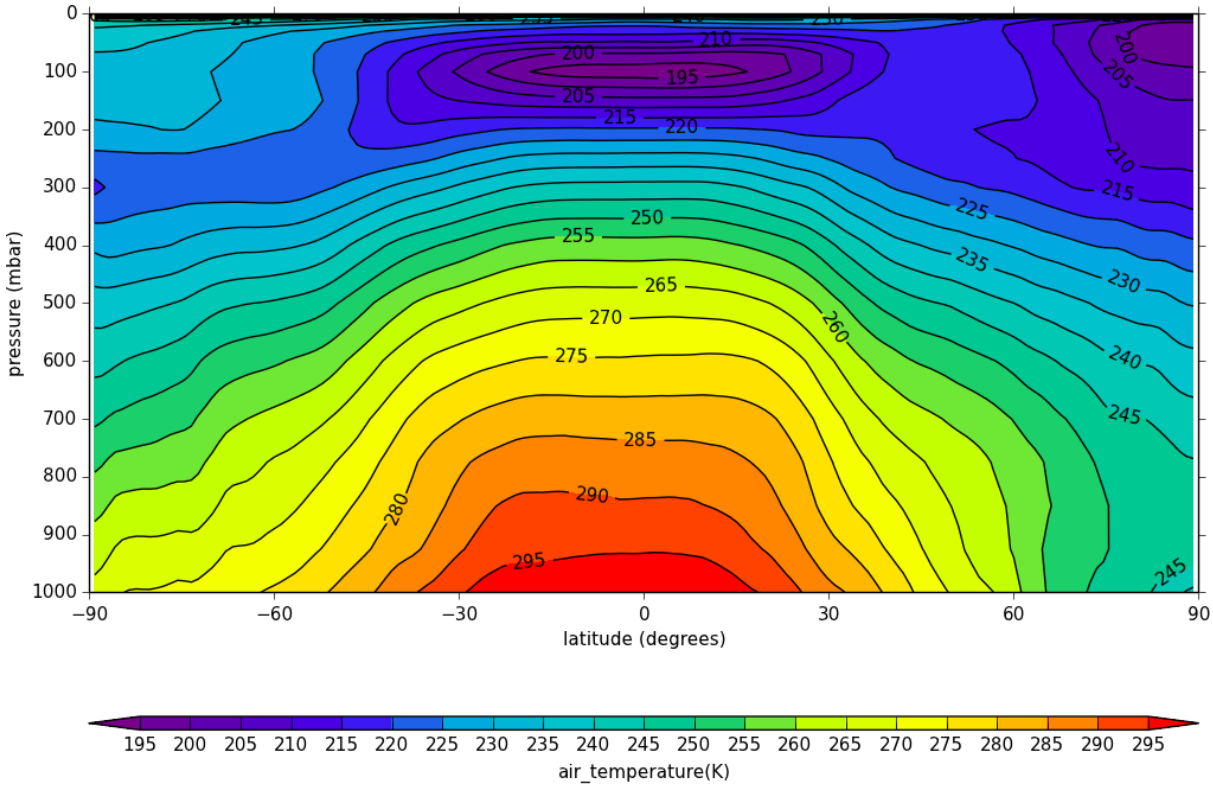


To mean the field use the cf collapse function:

cf.collapse(f[2], 'mean','longitude')

<CF Field: air_temperature(time(1), pressure(23), latitude(160), longitude(1)) K>

cfp.con(cf.collapse(f[2], 'mean','longitude'))



Using cf-plot with non-cf compliant data

Although newer model and reanalyis products are generally cf compliant there is a considerable amount of data being used that is of varying degrees of cf compliance.

In this locally processed ERA40 reanalysis field there is a distinct lack of standard names. Even the long names are somewhat terse - p for pressure for example.

f=cf.read('/opt/graphics/cfplot_data/ggap199006200600.nc')[9]

<CF Field: geopotential(long_name:t(1), long_name:p(37), long_name:latitude(256), long_name:longitude(512)) kg kg-1>**

We can still slice and plot the data by inspecting the dimensions of the field:

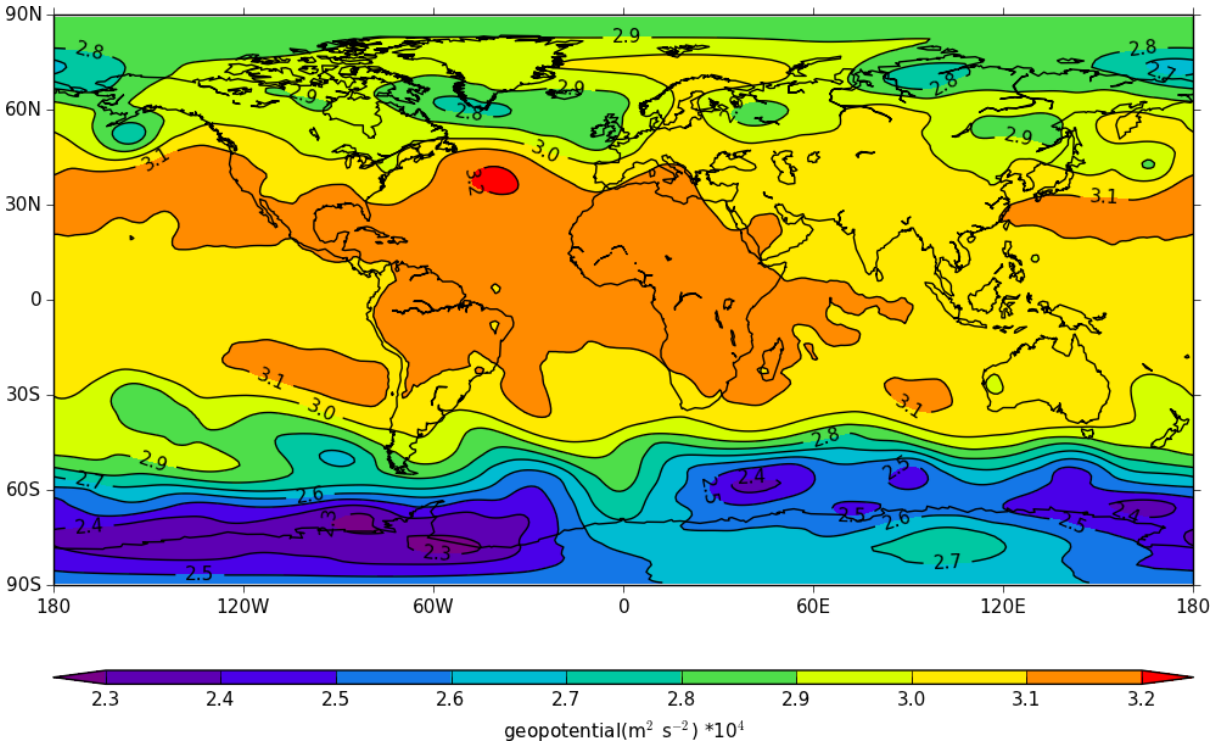
f.items()
{'dim2': <CF DimensionCoordinate: long_name:latitude(256) degrees_north>,
'dim3': <CF DimensionCoordinate: long_name:longitude(512) degrees_east>,
'dim0': <CF DimensionCoordinate: long_name:t(1) days since 1990-06-20 06:00:00>,
'dim1': <CF DimensionCoordinate: long_name:p(37) mbar>}

We see that dim1 is the pressure coordinate and can select the 700mb level and contour that.

f.item('dim1').array
array([1000., 975., 950., 925., 900., 875., 850., 825.,
800., 775., 750., 700., 650., 600., 550., 500.,
450., 400., 350., 300., 250., 225., 200., 175.,
150., 125., 100., 70., 50., 30., 20., 10.,
7., 5., 3., 2., 1.], dtype=float32)

f.subspace(dim1=700)
<CF Field: geopotential(long_name:t(1), long_name:p(1), long_name:latitude(256),
long_name:longitude(512)) kg kg-1>**

cfp.con(f.subspace(dim1=700))



In this dataset there are neither standard nor long names to identify the data.

f=cf.read('/opt/graphics/cfplot_data/gdata.nc')[0]
<CF Field: ncvar:temp(ncvar:p(22), ncvar:lat(73), ncvar:lon(145)) >

We can still slice and plot the data as below.

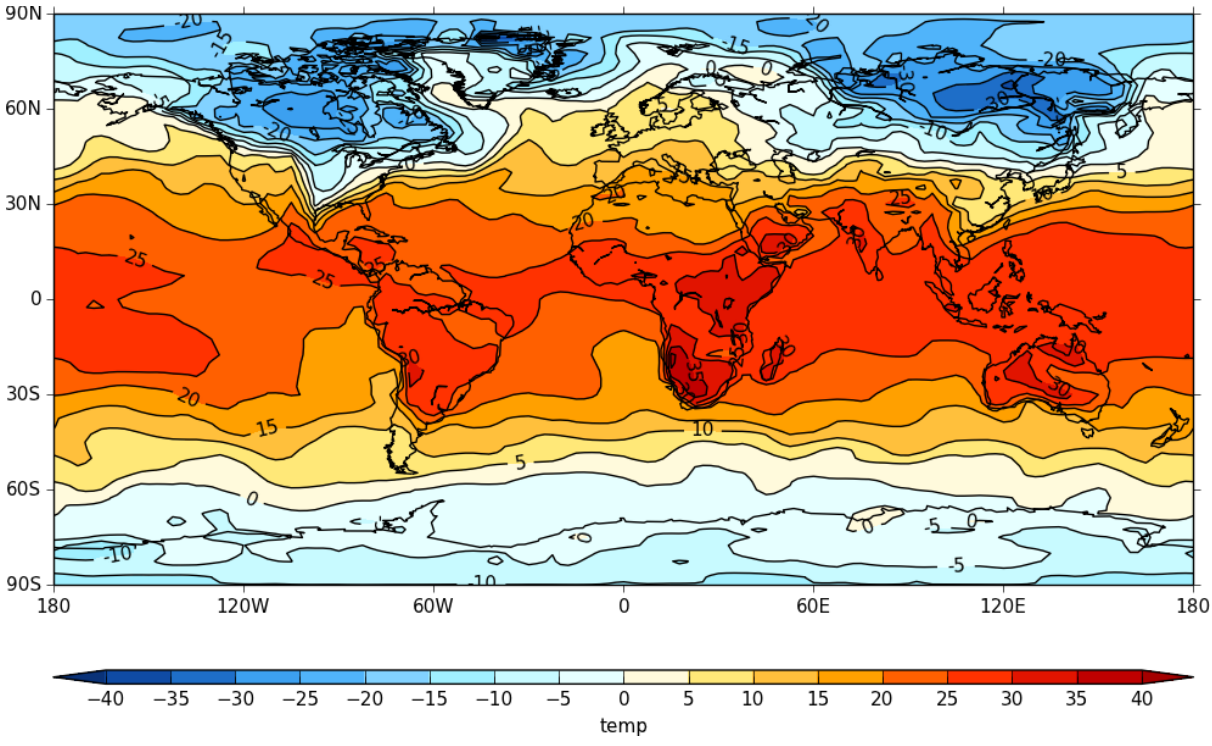
f.items()
{'dim2': <CF DimensionCoordinate: ncvar:lon(145)>,
'dim0': <CF DimensionCoordinate: ncvar:p(22)>,
'dim1': <CF DimensionCoordinate: ncvar:lat(73)>}

f.item('dim0').array

array([1.00000000e+03, 7.00000000e+02, 5.00000000e+02,
3.20000000e+02, 2.15000000e+02, 1.50000000e+02,
1.00000000e+02, 7.00000000e+01, 5.00000000e+01,
3.20000000e+01, 2.00000000e+01, 1.50000000e+01,
1.00000000e+01, 7.00000000e+00, 5.00000000e+00,
3.20000005e+00, 2.00000000e+00, 1.50000000e+00,
1.00000000e+00, 6.99999988e-01, 5.00000000e-01,
3.00000012e-01], dtype=float32)

So to plot the 1000mb temperature we would use:

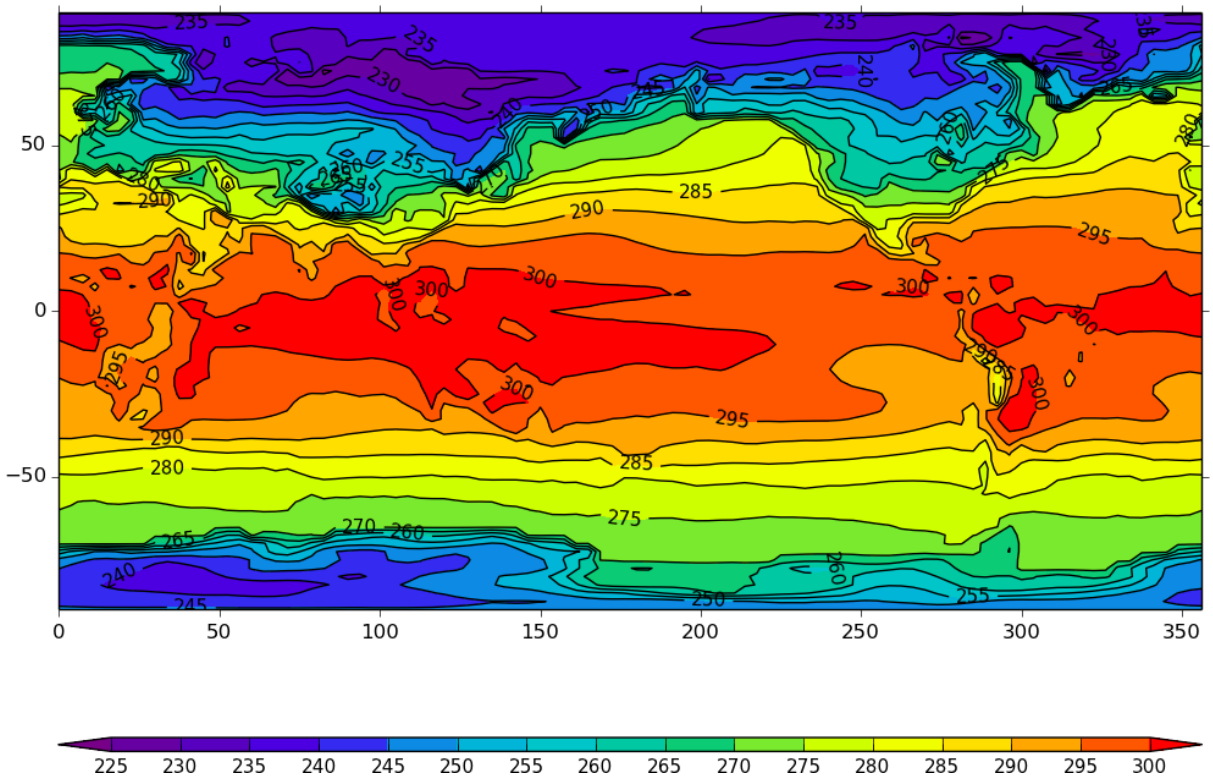
cfp.con(f.subspace(dim0=1000))



Passing data via arrays

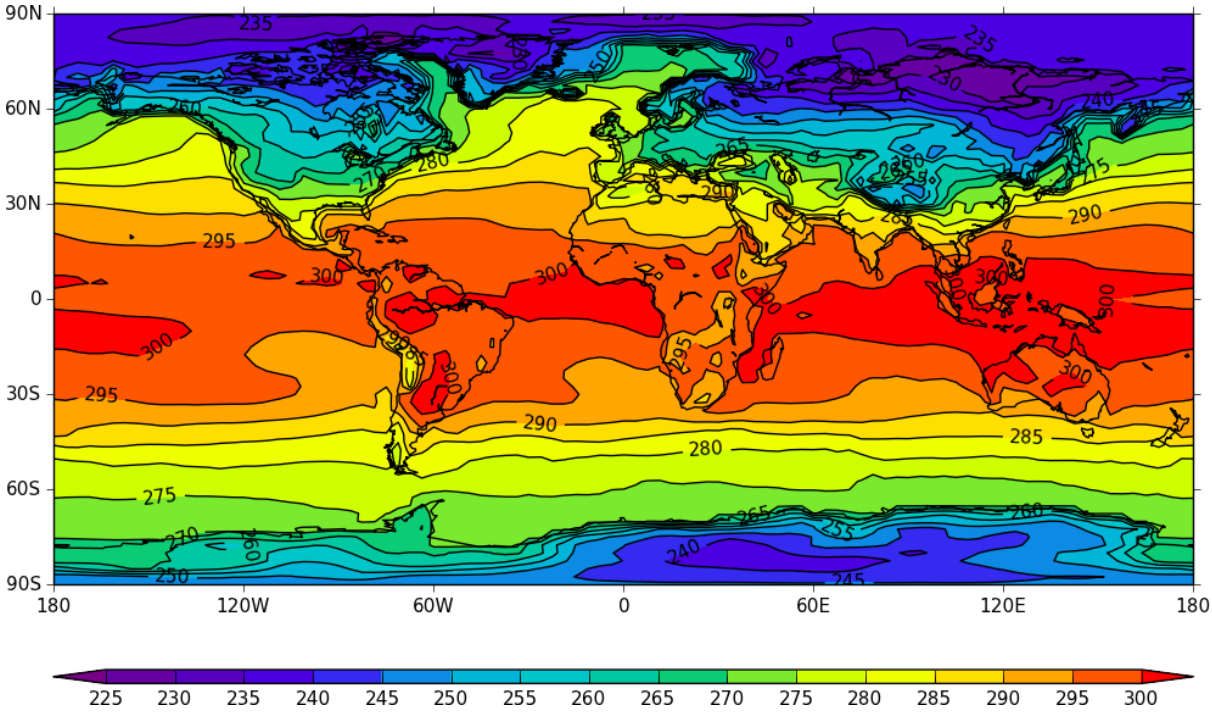
cfplot can also make contour and vector plots by passing data arrays. In this example we read in a temperature field from a netCDF file and pass it to cfplot for plotting.

```
import cfplot as cfp
from netCDF4 import Dataset as ncfile
import cfplot as cfp
nc = ncfile('/opt/graphics/cfplot_data/tas_A1.nc')
lons=nc.variables['lon'][:]
lats=nc.variables['lat'][:]
temp=nc.variables['tas'][0,:,:]
cfp.con(f=temp, x=lons, y=lats)
```



The contouring routine doesn't know that the data passed is a map plot. This can be explicitly set with the ptype flag to con.

cfp.con(f=temp, x=lons, y=lats, ptype=1)

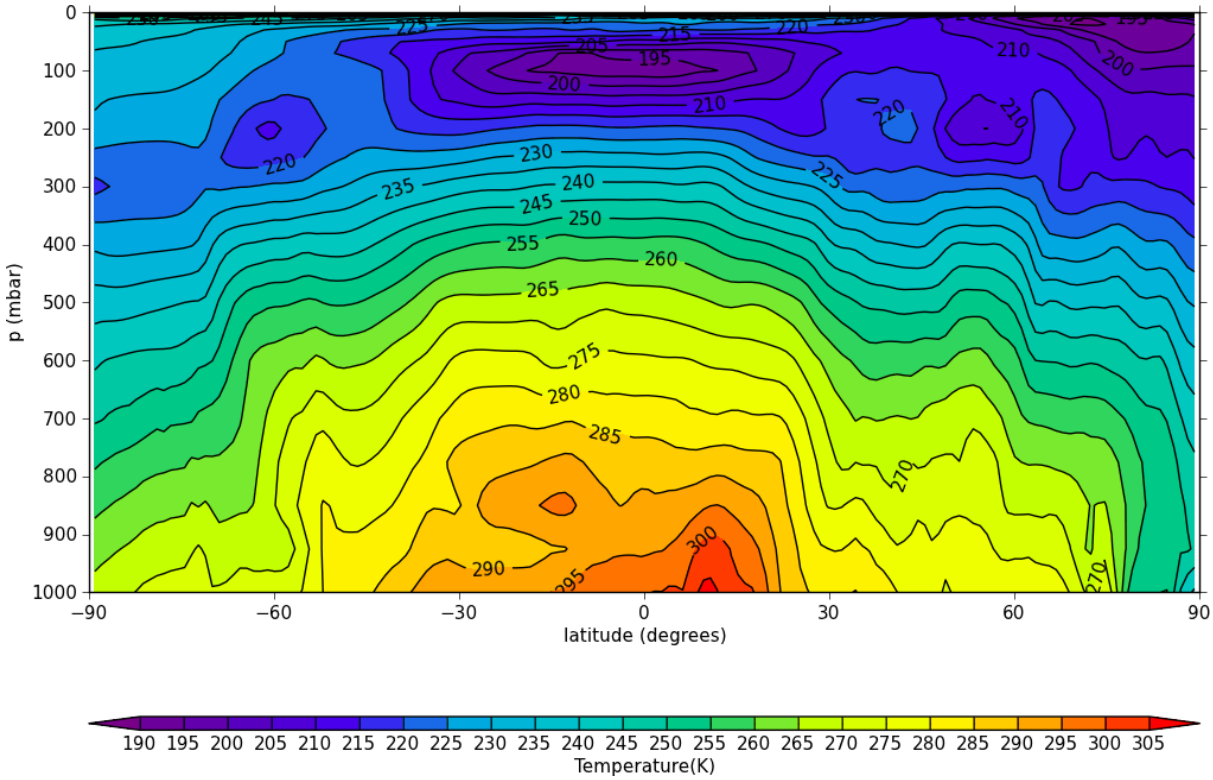


Other types of plot are:
ptype=2 - latitude - height plot
ptype=3 - longitude – height plot
ptype=4 - longitude - time plot
ptype=5 - latitude - time plot

Latitude - Pressure Plots

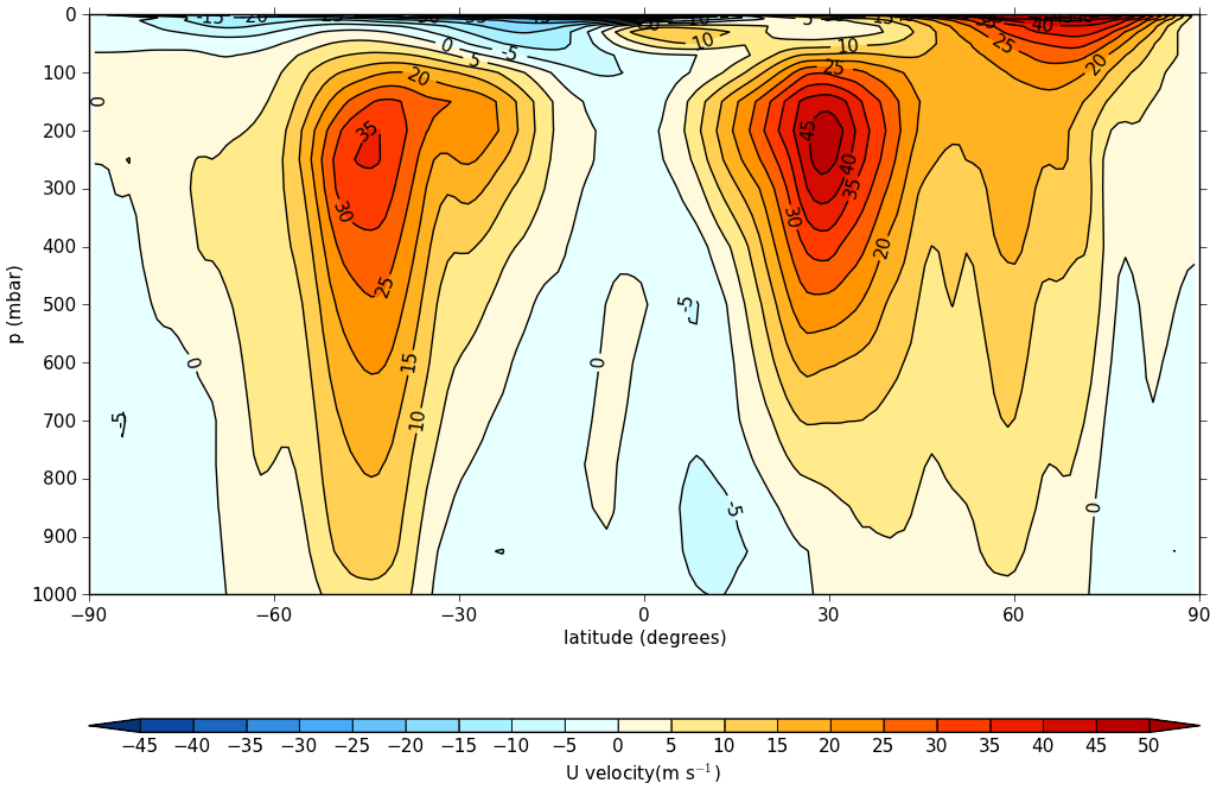
The latitude-pressure plot below is made by using the cf subspace method to select the temperature at longitude=0 degrees.

import cf, cfplot as cfp
f=cf.read('/opt/graphics/cfplot_data/ggap.nc')[2]
cfp.con(f.subspace(longitude=0))



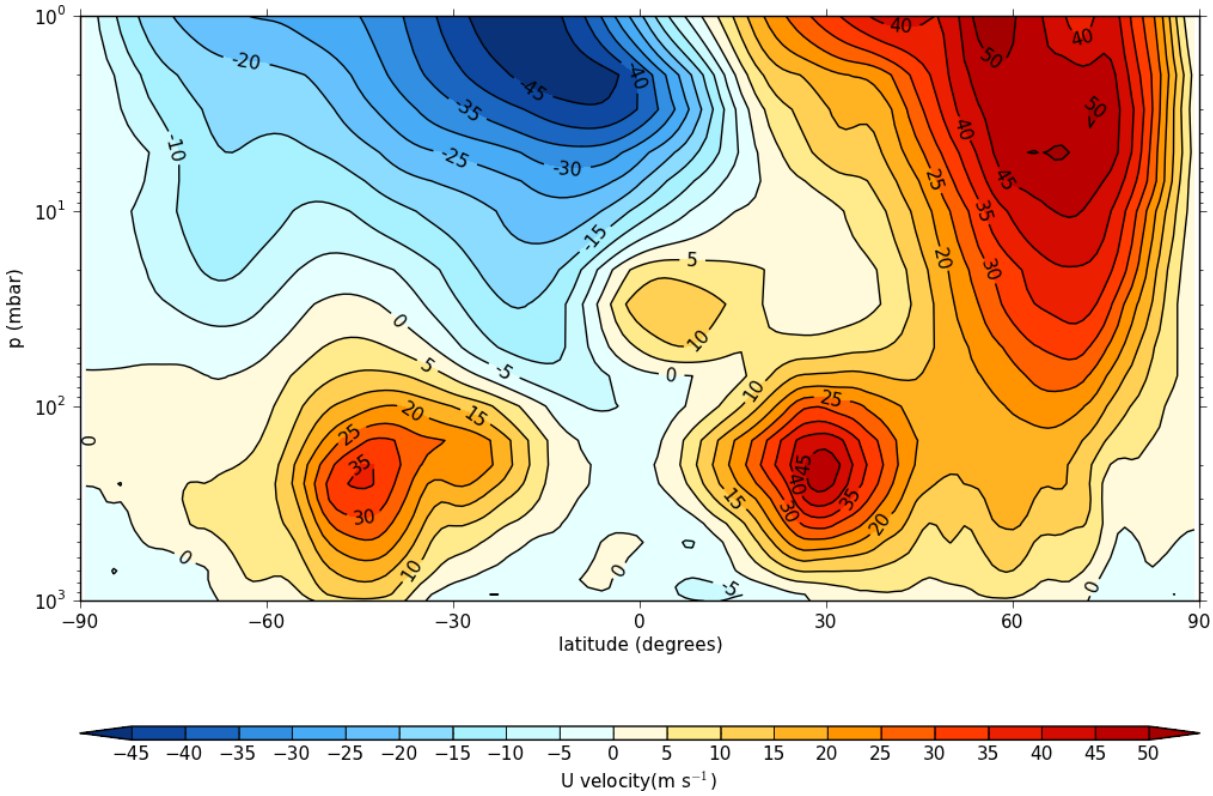
A mean of the data along the longitude (zonal mean) is made using the cf.collapse method.

```
import cf, cfplot as cfp
f=cf.read('/opt/graphics/cfplot_data/ggap.nc')[7]
cfp.con(cf.collapse(f, 'mean','longitude'))
```



To make a log pressure plot use **ylog=1** to the **con** routine.

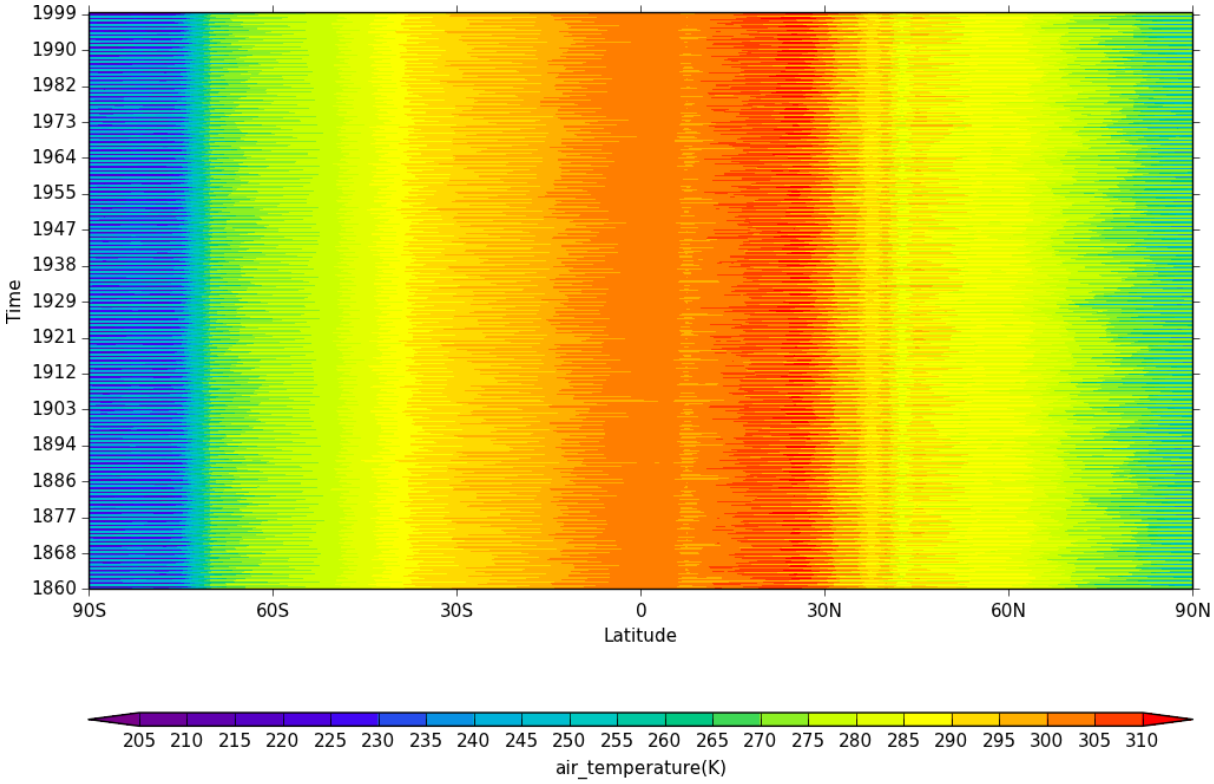
```
import cf, cfplot as cfp
f=cf.read('/opt/graphics/cfplot_data/ggap.nc')[7]
cfp.con(cf.collapse(f, 'mean','longitude'), ylog=1)
```



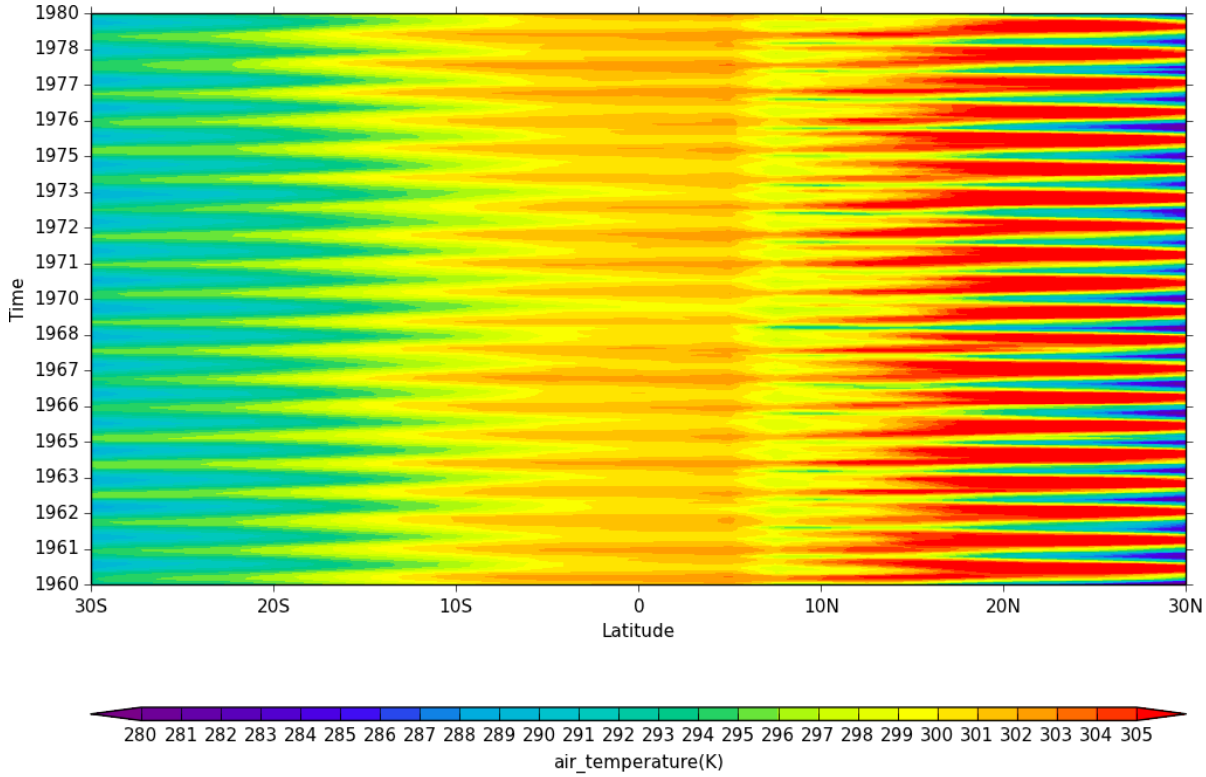
Hovmuller plots

A Hovmuller plot is one of longitude or latitude versus time as per the following examples.

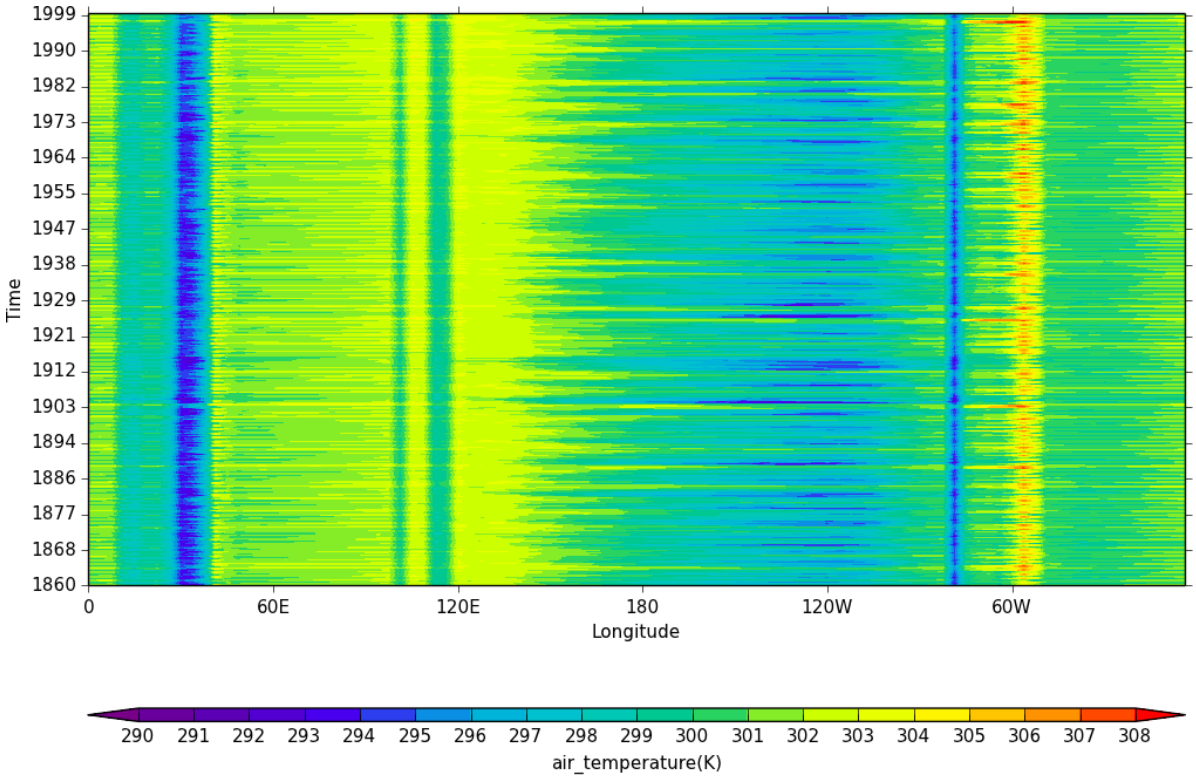
import cf, cfplot as cfp
f=cf.read('/opt/graphics/cfplot_data/tas_A1.nc')[0]
cfp.con(f.subspace(longitude=0), lines=0)



```
import cf, cfplot as cfp
f=cf.read('/opt/graphics/cfplot_data/tas_A1.nc')[0]
cfp.gset(-30, 30, '1960-1-1', '1980-1-1')
cfp.levs(min=280, max=305, step=1)
cfp.con(f.subspace(longitude=0), lines=0)
```



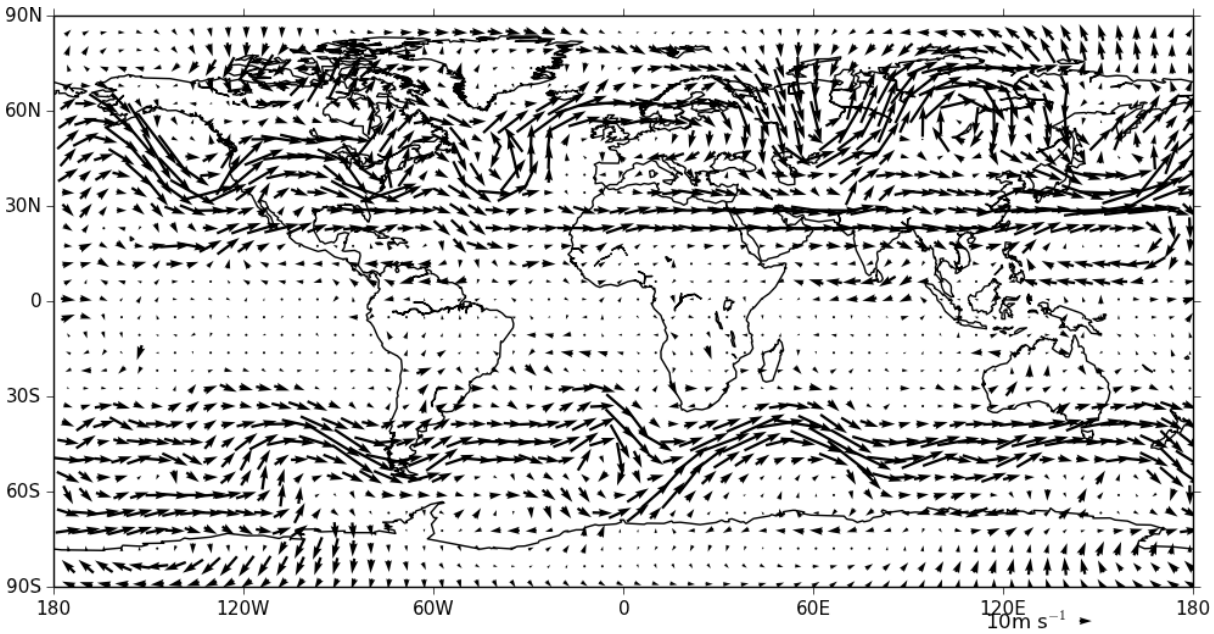
```
import cf, cfplot as cfp
f=cf.read('/opt/graphics/cfplot_data/tas_A1.nc')[0]
cfp.con(f.subspace(latitude=0), lines=0)
```



Vector Plots

Vector plots are made using the **vect** routine.

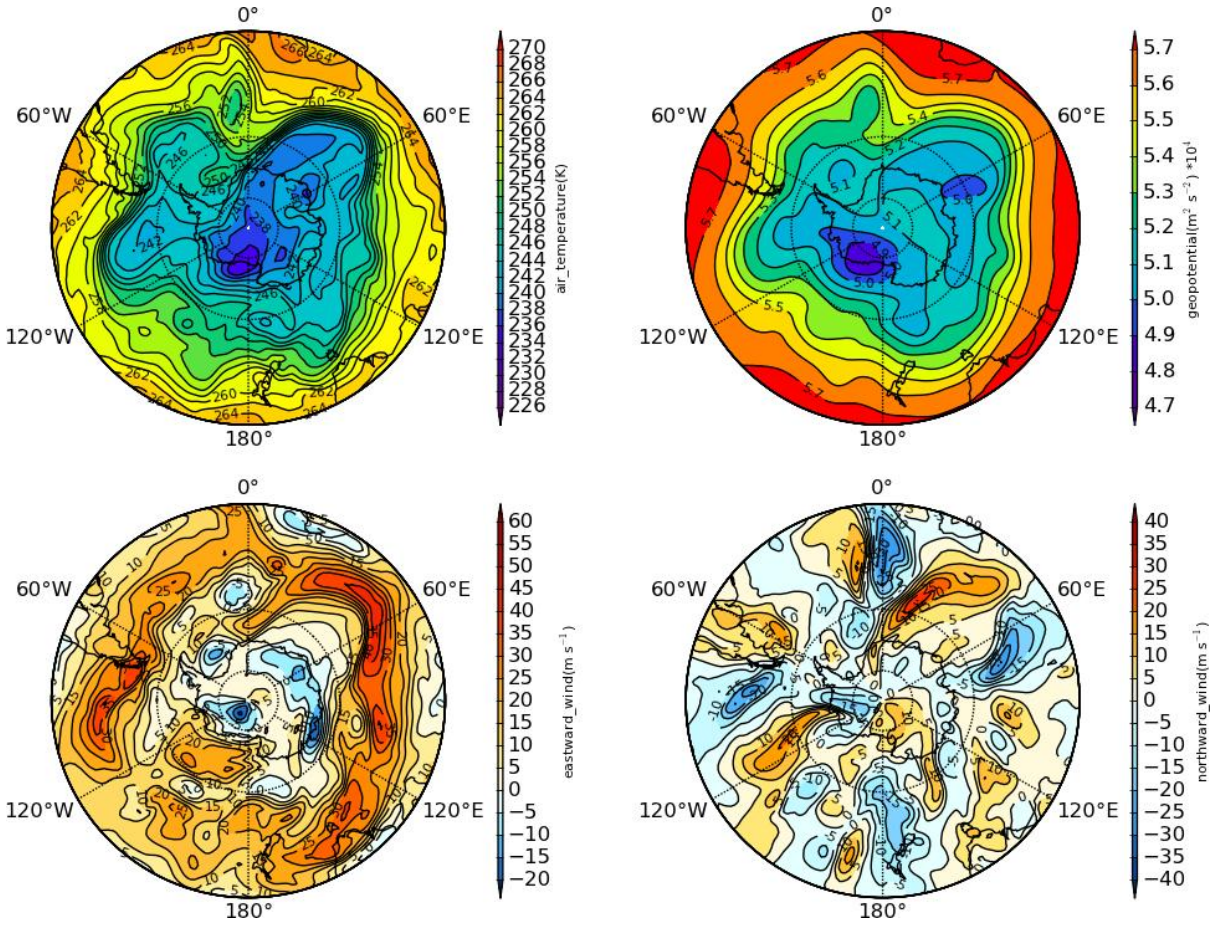
```
import cf, cfplot as cfp
f=cf.read('/opt/graphics/cfplot_data/ggap.nc')
u=f[7].subspace(pressure=500)
v=f[9].subspace(pressure=500)
cfp.vect(u=u, v=v, key_length=10, scale=100, stride=5)
```



Multiple Plots on a Page

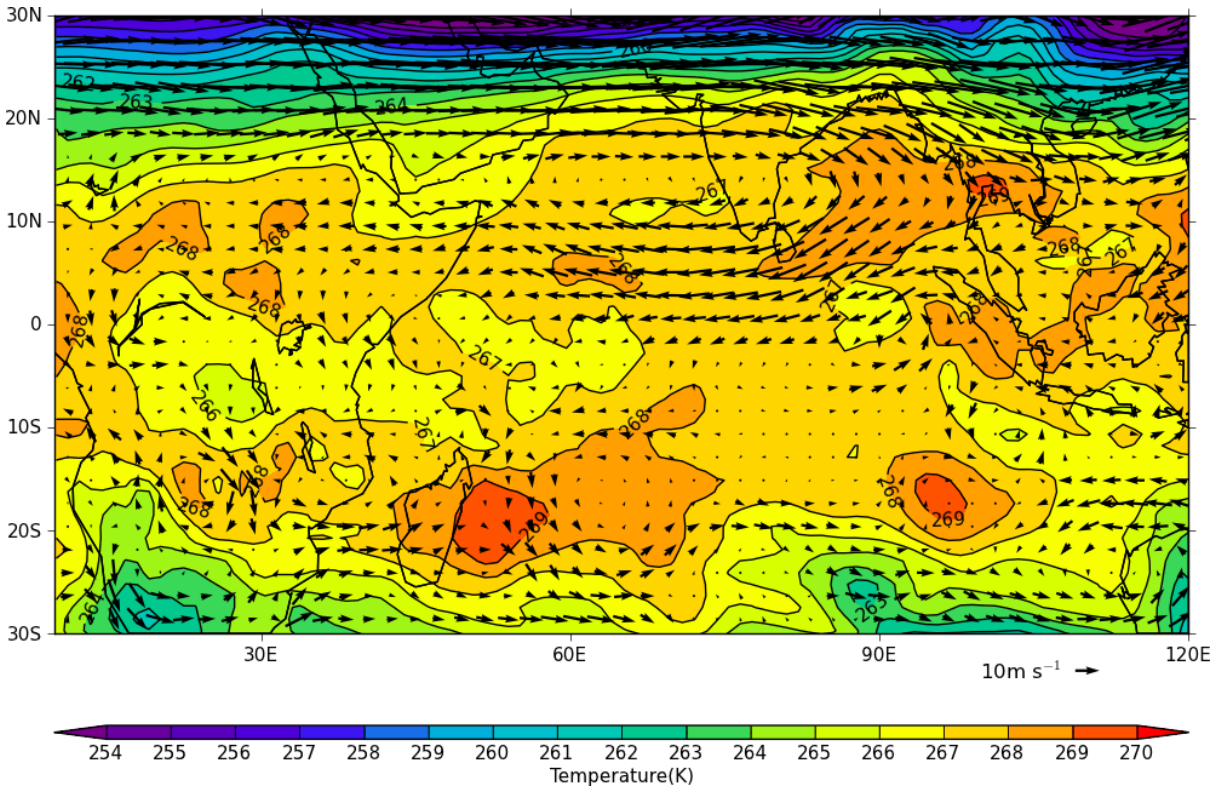
To make multiple plots on the page open a graphic file with the **gopen** command and pass the **rows** and **columns** parameters. Make each plot in turn first selecting the position with the **gpos** command. The first position is the top left plot and increases by one for one plot to the right until the final plot is made in the bottom right. When all the plots have been made close the plot with the **gclose()** command.

```
import cf, cfplot as cfp
f=cf.read('/opt/graphics/cfplot_data/ggap.nc')

cfp.gopen(rows=2, columns=2)
cfp.mapset(proj='spstere', boundinglat=-30, lon_0=180)
cfp.gpos(1)
cfp.con(f[2].subspace(pressure=500))
cfp.gpos(2)
cfp.con(f[8].subspace(pressure=500))
cfp.gpos(3)
cfp.con(f[7].subspace(pressure=500))
cfp.gpos(4)
cfp.con(f[9].subspace(pressure=500))
cfp.gclose()
```



In this example vectors are overlaid on a contour plot.

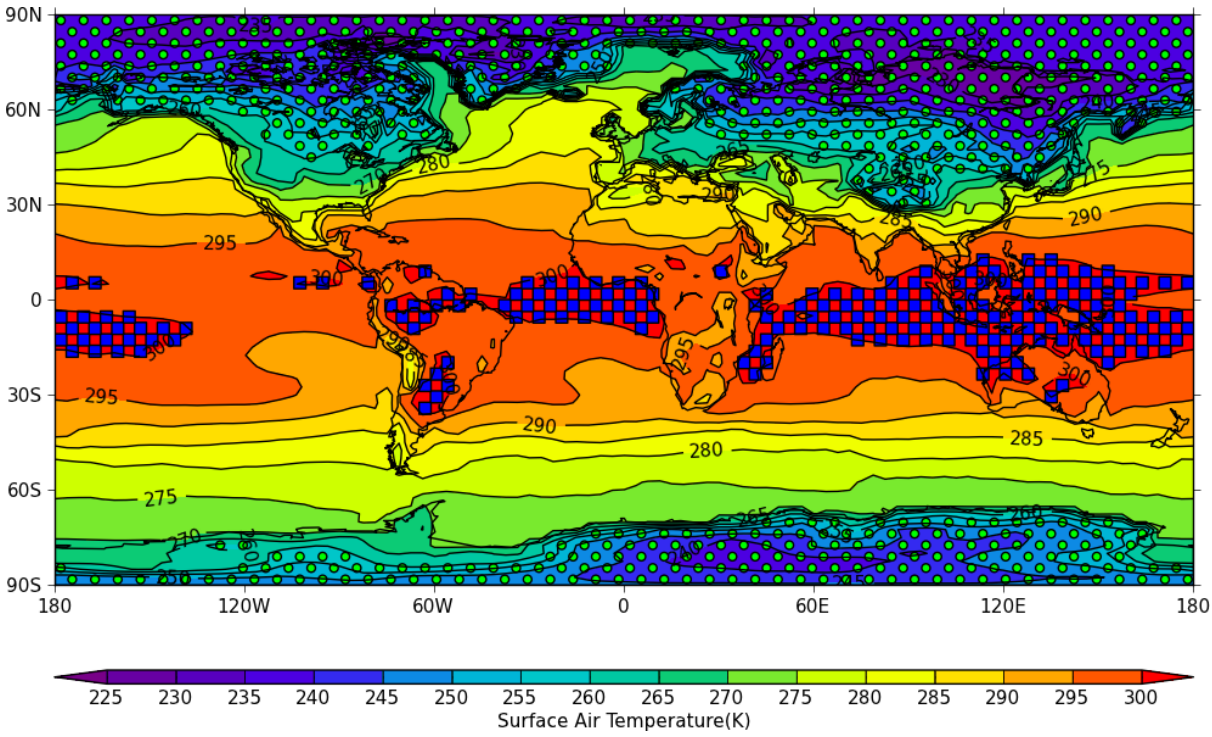
```
import cf, cfplot as cfp
f=cf.read('/opt/graphics/cfplot_data/ggap.nc')
u=f[7].subspace(pressure=500)
v=f[9].subspace(pressure=500)
t=f[2].subspace(pressure=500)

cfp.gopen()
cfp.mapset(lonmin=10, lonmax=120, latmin=-30, latmax=30)
cfp.levs(min=254, max=270, step=1)
cfp.con(t)
cfp.vect(u=u, v=v, key_length=10, scale=50, stride=2)
cfp.gclose()
```



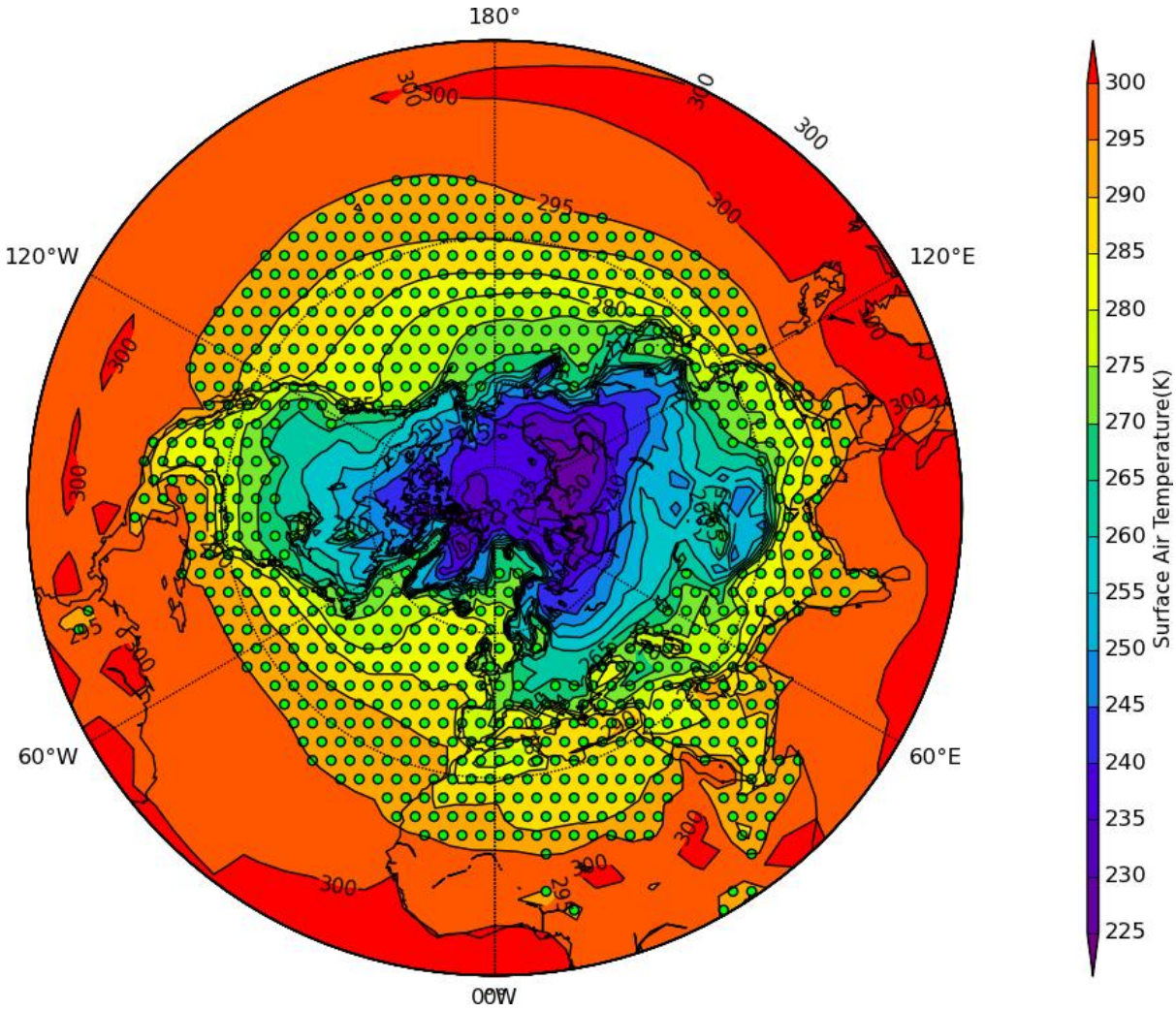
Stipple plots

A stipple plots is usually used to show areas of significance. These plots use the overlay technique as used in the previous contour/vector plot.

```
import cf, cfplot as cfp
f=cf.read('/opt/graphics/cfplot_data/tas_A1.nc')[0]
g=f.subspace(time=15)
cfp.gopen()
cfp.con(g)
cfp.stipple(f=g, min=220, max=260, size=100, color='#00ff00')
cfp.stipple(f=g, min=300, max=330, size=50, color='#0000ff', marker='s')
cfp.gclose()
```



```
import cf, cfplot as cfp
f=cf.read('/opt/graphics/cfplot_data/tas_A1.nc')[0]
g=f.subspace(time=15)
cfp.gopen()
cfp.mapset(proj='npstere')
cfp.con(g)
cfp.stipple(f=g, min=265, max=295, size=100, color='#00ff00')
cfp.gclose()
```



Setting Contour Levels

cfplot generally does a reasonable job of guessing appropriate contour levels. In the cases where it doesn't do this or you need a consistent set of levels between plots for comparison purposes use the **levs** routine.

The **levs** command manually sets the contour levels.
min=min - minimum level
max=max - maximum level
step=step - step between levels
manual= manual - set levels manually
extend='neither', 'both', 'min', or 'max' – the colour bar limit extensions. These are the triangles at the ends of the colour bar indicating the rest of the data is in this colour.

Use the **levs** command when a predefined set of levels is required. The **min**, **max** and **step** parameters are all needed to define a set of levels. These can take integer or floating point numbers. If colour filled contours are plotted then the default is to extend the minimum and maximum contours coloured for out of range values - **extend='both'**. Use the manual option to define a set of uneven contours i.e.
cfp.levs(manual=[-10, -5, -4, -3, -2, -1, 1, 2, 3, 4, 5, 10])

Once a user call is made to **levs** the levels are persistent. i.e. the next plot will use the same set of levels. Use **levs()** to reset to undefined levels i.e. let cfplot generate the levels again.

Once the **levs** command is used you'll need to think about the associated colour scale.

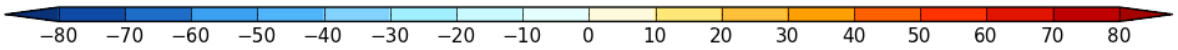
Colour scales

There are around 140 colour scales included with cfplot. Colour scales are set with the **cscale** command. There are two default colour scales that suit differing types of data.

A diverging scale ('scale1') that goes from blue to red and suits data with a zero in it. For example, temperature in Celsius or zonal wind.

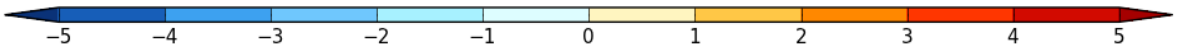
A continuous scale ('cosam') that goes from purple, blue, green, yellow, orange to red and suits data that has no zero in it. For example, air temperature in Kelvin or geopotential height.

cfp.levs(min=-80, max=80, step=10)
cfp.cscale('scale1')



To change the number of colours in a scale use the ncols parameters.

cfp.cscale('scale1', ncols=12)
cfp.levs(min=-5, max=5, step=1)



To change the number of colours above and below the mid-point of the scale use the above and below parameters. This is useful for fields where you have differing extents of data above and below the zero line.

cfp.cscale('scale1', below=4, above=7)
cfp.levs(min=-30, max=60, step=10)

For data where you need white to indicate that this data region is insignificant use the white=white parameter. This can take single or multiple values.

cfp.cscale('scale1', ncols=11, white=5)
cfp.levs(manual=[-10,-8, -6, -4, -2, 2, 4, 6, 8, 10])

User defined colour scales

Store these as rgb values in a file with one rgb value per line. i.e.

255 0 0
255 255 255
0 0 255

will give a red white blue colour scale. If the file is saved as /home/andy/rwb.txt it is read in using

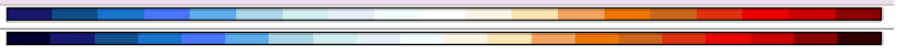
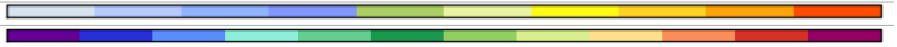
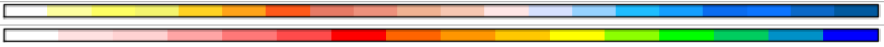
cfp.cscale('/home/andy/rwb.txt')

If the colour scale selected has too few colours for the number of contour levels then the colours will be used cyclically.

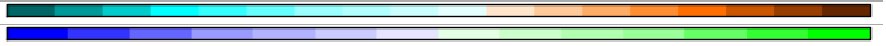
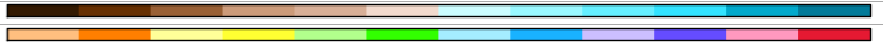
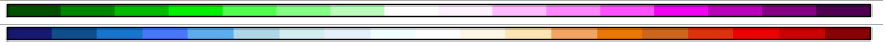
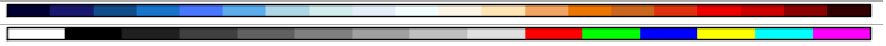
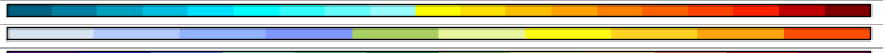
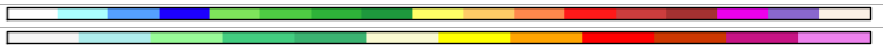
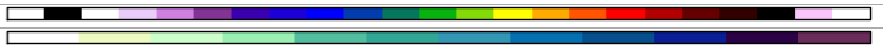
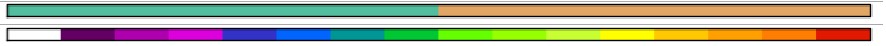
Predefined colour scales

A lot of the following colour maps were downloaded from the NCAR Command Language web site. Users of the IDL guide colour maps can see these maps at the end of the colour scales.

NCAR Command Language - MeteoSwiss colour maps

Name	Scale
hotcold_18lev	
hotcolr_19lev	
mch_default	
perc2_9lev	
percent_11lev	
precip2_15lev	
precip2_17lev	
precip3_16lev	
precip4_11lev	
precip4_diff_19lev	
precip_11lev	
precip_diff_12lev	
precip_diff_1lev	
rh_19lev	
spread_15lev	

NCAR Command Language - small color maps (<50 colours)

Name	Scale
amwg	
amwg_blueyellowred	
BlueDarkRed18	
BlueDarkOrange18	
BlueGreen14	
BrownBlue12	
Cat12	
cmp_flux	
cosam12	
cosam	
GHRSST_anomaly	
GreenMagenta16	
hotcold_18lev	
hotcolr_19lev	
mch_default	
nrl_sirkes	
nrl_sirkes_nowhite	
perc2_9lev	
percent_11lev	
posneg_2	
prcp_1	
prcp_2	
prcp_3	
precip_11lev	
precip_diff_12lev	
precip_diff_1lev	
precip2_15lev	
precip2_17lev	
precip3_16lev	
precip4_11lev	
precip4_diff_19lev	

radar_1	
rh_19lev	
seaice_1	
seaice_2	
so4_21	
spread_15lev	
StepSeq25	
sunshine_9lev	
sunshine_diff_12lev	
temp_19lev	
temp_diff_18lev	
temp_diff_1lev	
topo_15lev	
wgne15	
wind_17lev	

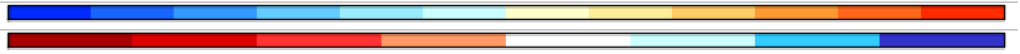
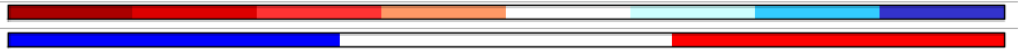
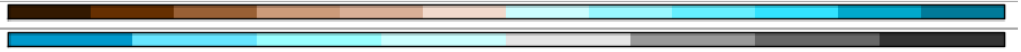
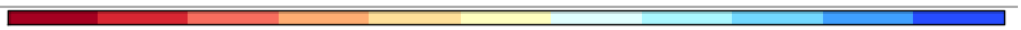
NCAR Command Language - large colour maps (>50 colours)

Name	Scale
amwg256	
BkBlAqGrYeOrReViWh200	
BlAqGrYeOrRe	
BlAqGrYeOrReVi200	
BlGrYeOrReVi200	
BlRe	
BlueRed	
BlueRedGray	
BlueWhiteOrangeRed	
BlueYellowRed	
BlWhRe	
cmp_b2r	
cmp_haxby	
detail	
extrema	
GrayWhiteGray	
GreenYellow	
helix	
helix1	
hotres	
matlab_hot	
matlab_hsv	
matlab_jet	
matlab_lines	
ncl_default	
ncview_default	
OceanLakeLandSnow	
rainbow	
rainbow+white+gray	
rainbow+white	
rainbow+gray	
tbr_240-300	
tbr_stdev_0-30	
tbr_var_0-500	
tbrAvg1	
tbrStd1	
tbrVar1	
thelix	
ViBlGrWhYeOrRe	
wh-bl-gr-ye-re	
WhBlGrYeRe	
WhBlReWh	
WhiteBlue	
WhiteBlueGreenYellowRed	
WhiteGreen	
WhiteYellowOrangeRed	
WhViBlGrYeOrRe	
WhViBlGrYeOrReWh	
wxpEnIR	
3gauss	
3saw	

NCAR Command Language - Enhanced to help with colour blindness

Name	Scale
StepSeq25	
posneg_2	
posneg_1	
BlueDarkOrange18	
BlueDarkRed18	
GreenMagenta16	
BlueGreen14	
BrownBlue12	
Cat12	

IDL guide scales

Name	Scale
scale1	
scale2	
scale3	
scale4	
scale5	
scale6	
scale7	
scale8	
scale9	
scale10	
scale11	
scale12	
scale13	
scale14	
scale15	
scale16	
scale17	
scale18	
scale19	
scale20	
scale21	
scale22	
scale23	
scale24	
scale25	
scale26	
scale27	
scale28	
scale29	
scale30	
scale31	
scale32	
scale33	
scale34	
scale35	
scale36	
scale37	
scale38	
scale39	
scale40	
scale41	
scale42	
scale43	
scale44	

Making postscript or PNG pictures

There are two methods of producing a figure for use in an external package such as a web document or LaTeX, Word etc.

cfp.plotvars.file='zonal.ps' write graphics output to a file called zonal.ps
cfp.plotvars.file='zonal.png' write graphics output to a file called zonal.png

To reset to viewing the picture on the screen again use **cfp.plotvars.file=None.** Note the None here is not in quotes.

The second way is to use **gopen** as when used in making multiple plots.

gopen, file='zonal.ps'
cfp.con(f.subspace(time=15))
gclose()

Making non-blocking plots

Using the Python prompt a cfplot plot on the screen will block further interaction vial the prompt. To get around this start Python with **ipython –pylab**

Using cfplot in batch mode

The following method works in the Reading Meteorology department.

In the file /home/andy/ajh.sh:
#!/bin/sh
/home/opt-user/Enthought/Canopy_64bit/User/bin/python /home/andy/ajh.py

In the file /home/andy/ajh.py:
import cf
import cfplot as cfp
f=cf.read('/opt/graphics/cfplot_data/tas_A1.nc')[0]
cfp.plotvars.file='/home/andy/ll.png'
cfp.con(f.subspace(time=15))

run the batch job at 16:33:

at -f /home/andy/ajh.sh 16:33

Python for IDL users

This section is a guide for IDL users using Python for the first time. These notes are intended to help get you started using Python and making graph plots with Matplotlib.

A more detailed look at Python is available on-line at Johnny Lin's website.

Importing packages

In IDL all the functionality of the language is available by just typing idl - maps, stats etc. In Python you have to explicitly import packages you need. Some of the commonly used packages are:

import numpy as np - scientific computing
from scipy import interpolate – interpolation
import matplotlib.pyplot as plt – plotting
from mpl_toolkits.basemap import Basemap, shiftgrid, addcyclic – mapping
from subprocess import call - calling Unix programs
from netCDF4 import Dataset as ncfile - netCDF I/O
import cf - David Hassell's cf data I/O

After importing a package functions can then referenced - for example, **np.mean(a)** for the NumPy mean function.

Running Python programs

In Python no end statement or procedure/function name is needed. To run a Python program type

python prog.py

Python basics

Python floats are at least 64 bit numbers.

Python integers are 32 bit and not 16 bit as in IDL. A long integer is specified by putting a L or l after the number. Long integers can grow as large as is needed.

In Python variable names case sensitive while in IDL they are case insensitive. There is no need to declare variable types - just use them. As in IDL you can reassign variables to be different types.

is the comment symbol. All text after a # is a comment.

Use **print a,b,c** to see the variables a,b,c. In IDL this would be print, a,b,c.

Array elements are accessed via [] brackets only. In IDL you can use both [] and (). Both Python and IDL indices start at zero - i.e. the first element is a[0].

Indentation is the way of closing if then statements or loops in Python - there's no need for an endif or endfor as in IDL.

The rules of operator precedence are the same as in IDL - an integer multiplied by a float gives a float.

Python has a C style data interface while IDL has a Fortran style. If we have read in a standard atmosphere grid, in IDL it will appear as longitude, latitude, height, in Python it will appear as height, latitude, longitude. In IDL you would use a[i,*], the equivalent in Python is a[:,i]. See section on netCDF reading later.

Loops in Python

```
for i in np.arange(4):
  print i
```

```
0
1
2
3
```

Python is indentation sensitive so make sure your loop statements are indented. There is no end of loop statement in Python.

Large loops in Python are slow. The following example is ten times slower than in IDL for a loop of 100 million points. A standard UM grid 96x73x22 for 360 days has 57 million points so this is not an unusual amount of data to be looking at.

First of all we create some test random data from -5 to 5. We will loop over the data and set the b array to be 1 for all points that are greater than zero.

```
a=np.random.rand(100000000)*10-5
b=np.zeros(100000000,dtype=int)
pts=np.arange(0,100000000)
for i in pts:
  if a[i] > 0: b[i]=1
```

This took 160 seconds for a 100 million points. Using **pts=range(0,100000000)** reduces this to 100 seconds.

Using NumPy **where** gives a much quicker answer **b=np.where(a < 0, a, 1)** took 3.1 seconds

As a comparison using WHERE in IDL completed the same task in 0.16 seconds.

Creating arrays

The NumPy arange function is like INDGEN, FINDGEN in IDL. Up to three arguments are generally used - min, max and step. Note that result doesn't include the max value passed to it.

np.arange(5) - array([0, 1, 2, 3, 4])
np.arange(2,6) - array([2, 3, 4, 5])
np.arange(3,9,1.5) - array([3., 4.5, 6., 7.5])

Selecting data

a=np.arange(0,10,1) - array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
a[0] – 0
a[-1] – 10
a[5:] - array([5, 6, 7, 8, 9, 10])
a[:5] - array([0, 1, 2, 3, 4])

The colon delimiter is used to indicate the beginning or end of a sequence. When used singly it means all values.

If / else examples

There are a variety of ways of doing an if statement in Python.

```
if condition:
  statements
else:
  statements
```

Python is indentation sensitive so make sure your statements are indented. There is no end of if statement in Python.

One line if commands are as follows:

```
if condition: do_something()
if condition: do_something(); do_something_else()
```

The following is also valid but more difficult to work out the logic than a multi-line equivalent.

```
a = 1 if x > 15 else 2
```

Python conditionals

a == b Equal
a < b Less than

a > b Greater than
a <= b Less than or equal
a >= b Greater than or equal
a != b Not Equal

NumPy and Scipy documentation

The top level NumPy documentation is at http://www.numpy.org
Scipy routine list is at http://docs.scipy.org/doc/scipy/reference/index.html

For FFTs, regressions etc. look in the SciPy documenation.

Useful NumPy expressions

np.sqrt(a) - square root
np.log(a) - logarithm, base e (natural)
np.log10(a) - logarithm, base 10
np.exp(a) - exponential function

np.max(a), **np.min(a)**, **np.mean(a)**, **np.shape(a)**, **np.ndim(a)** - max, min, mean, shape, number of dimensions of a. These also take the **axis=axis** keyword to do function along an axis. The first axis is **axis=0**.

np.pi – pin
p.nan - not a number
np.inf - infinity

To create a 3,5 array of np.zeros((3,5))+5

Averaging over axes:

np.mean(temp, axis=2) - note that axis=0 is the first axis
np.sum(temp, axis=1) – sum over an axis
np.ceil(a) - round up
np.floor(a) - round down
np.int(a), **np.float(a)** - convert to integer, float
np.arange(5) - array([0, 1, 2, 3, 4])

Calendar functions

These are available via the netcdftime module.

```
from netcdftime import utime
from datetime import datetime
cdftime = utime('hours since 0001-01-01 00:00:00')
print cdftime.units,'since',cdftime.origin
print cdftime.calendar,'calendar'
d = datetime(2006,9,29,12)
```

print cdftime.date2num(d)
print cdftime.num2date(t1)

hours since 1-01-01 00:00:00
standard calendar
17582028.0
2006-09-29 12:00:00

Passing data to routines and getting results back.

Define a routine with a def statement.

```
def myproc(min=min, max=max):
   return max-min
```

Don't forget to indent the lines after the procedure **def** line.

myproc(min=1, max=4) – 3
myproc(max=4, min=1) - 3

myproc(1,4) - 3 - this is using Python's keyword position to avoid putting in the min and max keywords.

To return multiple values return them comma separated with the return command

```
def myproc(min=min, max=max):
  return max-min, min+(max-min)/2.0
```

myproc(min=1, max=4) - (3, 2.5)

a,b=myproc(min=1, max=4)

Reading in netCDF data

Use the netCDF4 python interface.

```
from netCDF4 import Dataset as ncfile
nc=ncfile('gdata.nc')
lons=nc.variables['lon'][:]
lats=nc.variables['lat'][:]
p=nc.variables['p'][:]

temp=nc.variables['temp'][0,:,:]
```

: is a delimeter which here means all values. Python orders its data C style and IDL Fortran style. temp is a lon,lat,height grid and that's how it would appear in IDL. In Python it appears as height,lat,lon. In this example temp=nc.variables['temp'][0,:,:] gives the first value of pressure - 1000mb and all the longitudes and latitudes.

Timing functions

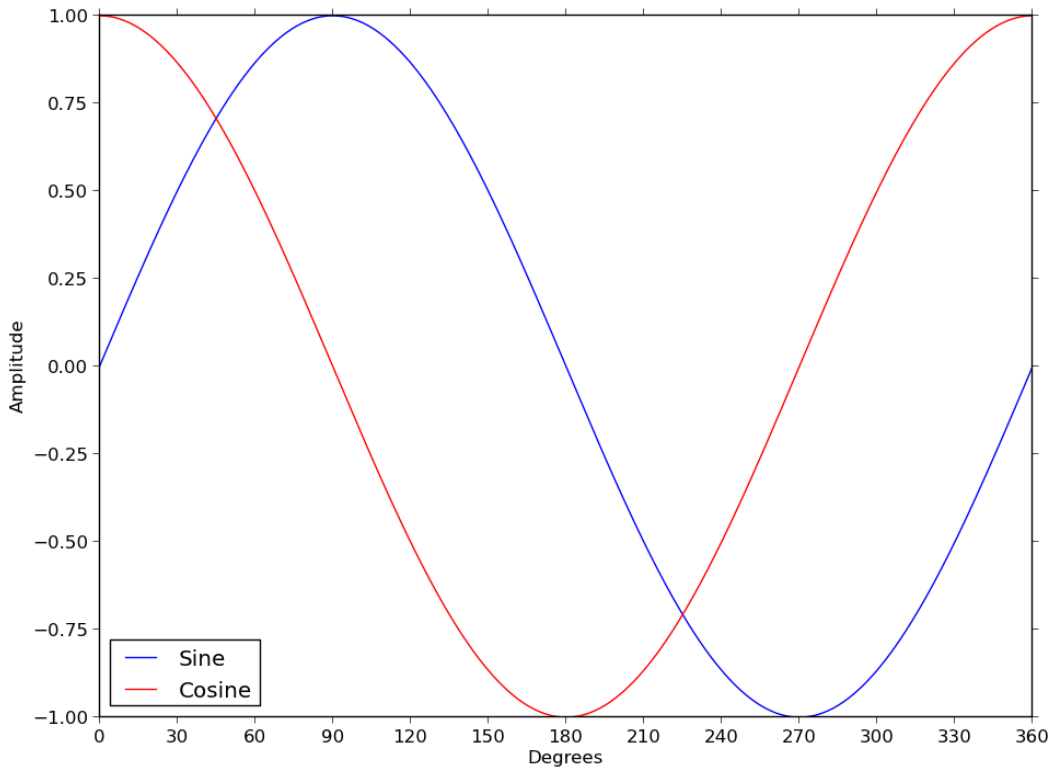
To see where your code is spending the most time use the time module

```
import time
import numpy as np
t1 = time.time()
for i in  np.arange(1e5):
  a=np.cos(i)
t2 = time.time()
print 'Time taken was ', t2-t1, 'seconds'
```

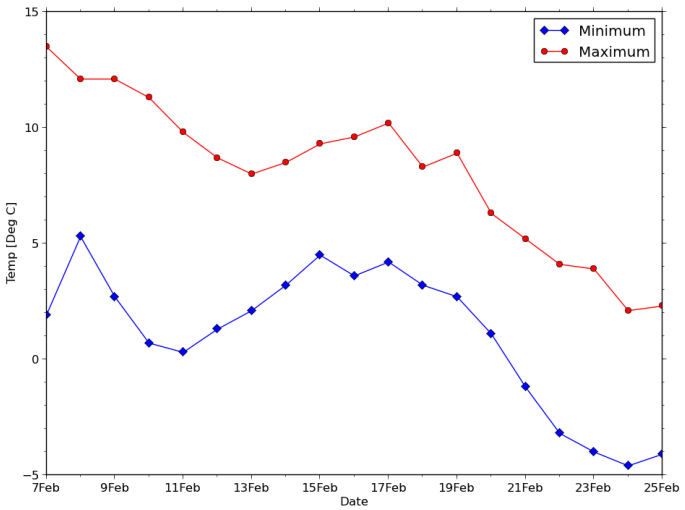
Making graphs in Python

A series of common graphs and the code used to make them

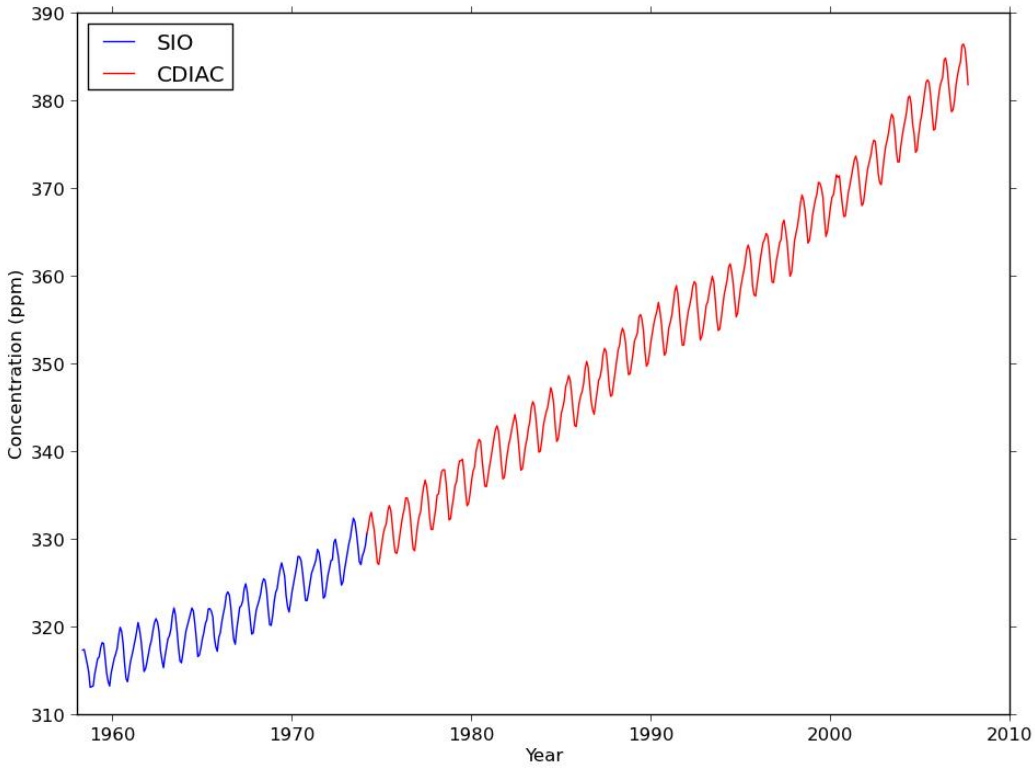
```
#!/usr/bin/env python
import matplotlib.pyplot as plt
import numpy as np

fig = plt.figure(figsize=(11, 8))
ax1 = fig.add_subplot(111)

ax1.axis([0, 360, -1, 1])
ax1.tick_params(direction='out', which='both')
ax1.set_xlabel('Degrees')
ax1.set_ylabel('Amplitude')
ax1.set_xticks(np.arange(0, 361, 30))
ax1.set_yticks(np.arange(-1, 1.1, 0.25))

xpts=np.arange(0,361,1)
ax1.plot(xpts, np.sin(np.radians(xpts)), label='Sine', color="blue")
ax1.plot(xpts, np.cos(np.radians(xpts)), label='Cosine', color="red")

ax1.legend(loc='lower left')

plt.savefig('ex35.png')
```



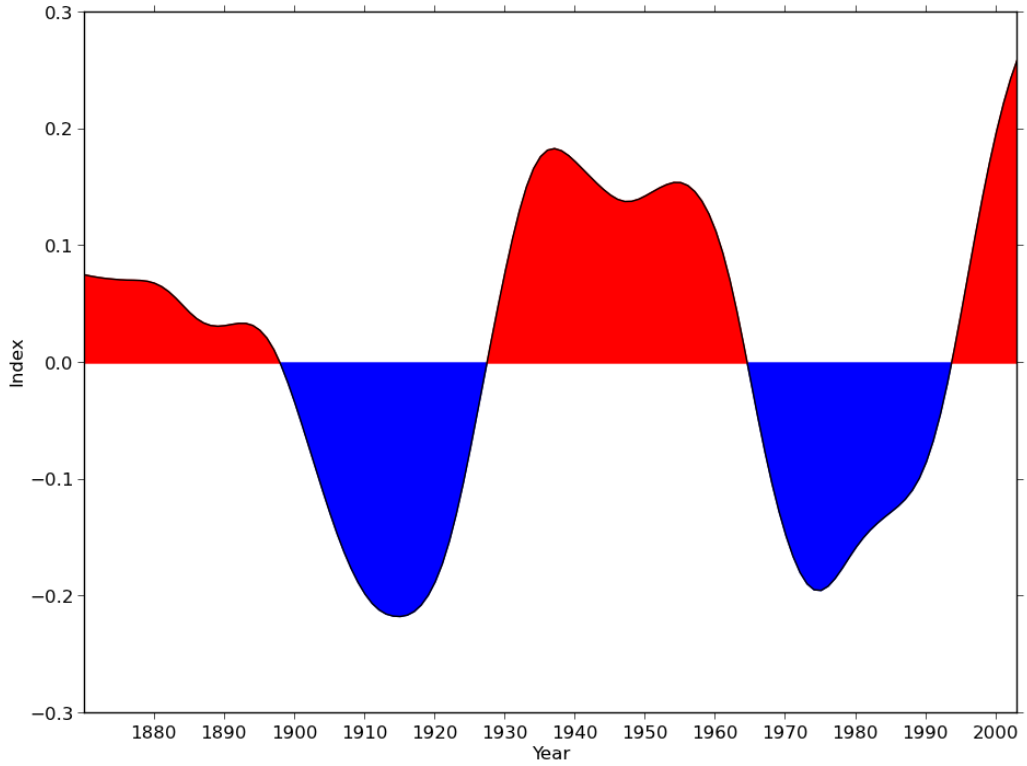
```
#!/usr/bin/env python
import matplotlib.pyplot as plt
import numpy as np
from netCDF4 import Dataset as ncfile
from matplotlib.ticker import MultipleLocator

nc = ncfile('/opt/graphics/cfplot_data/fieldsite.nc')
days=nc.variables['day'][:]
temp=nc.variables['temperature'][:]

fig = plt.figure(figsize=(11, 8))
ax1 = fig.add_subplot(111)

ax1.axis([7, 25, -5, 15])
ax1.tick_params(direction='out', which='both')
ax1.tick_params(bottom='on')
ax1.set_xlabel('Date')
ax1.set_ylabel('Temp [Deg C]')
ax1.set_xticks(np.arange(7,26,2))

xlabels=[]
for i in np.arange(7,26,2): xlabels.append(str(i)+'Feb')
ax1.set_xticklabels(xlabels)
ax1.set_yticks(np.arange(-5, 16, 5))
ax1.xaxis.set_minor_locator(MultipleLocator(1))
ax1.yaxis.set_minor_locator(MultipleLocator(1))
ax1.plot(days, temp[0,:], label='Minimum', color="blue", marker='D')
ax1.plot(days, temp[1,:], label='Maximum', color="red", marker='o')

ax1.legend(loc='upper right')

plt.savefig('ex36.png')
```



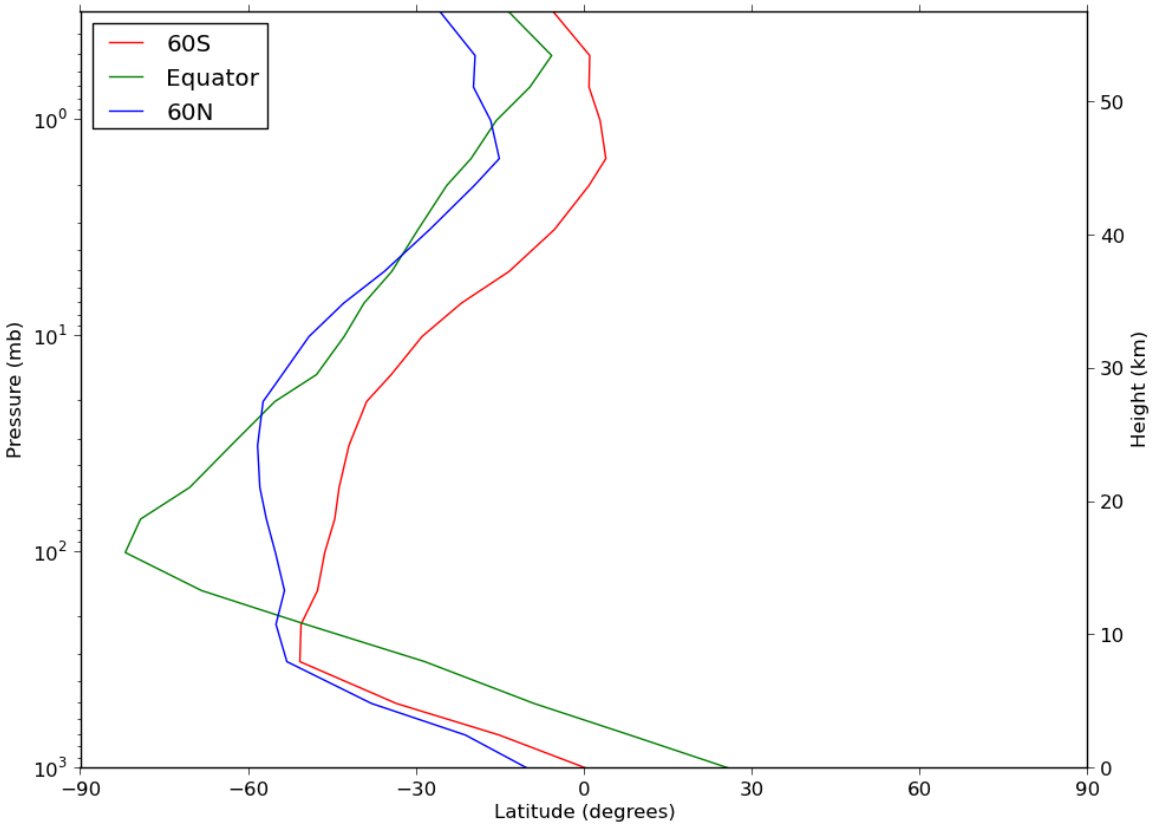
```
!/usr/bin/env python
import matplotlib.pyplot as plt
import numpy as np
from netCDF4 import Dataset as ncfile

nc = ncfile('/opt/graphics/cfplot_data/maunaloa.nc')
time=nc.variables['time'][:]
co2=nc.variables['co2'][:]

fig = plt.figure(figsize=(11, 8))
ax1 = fig.add_subplot(111)

ax1.axis([1958, 2010, 310, 390])
ax1.tick_params(direction='out', which='both')
ax1.set_xlabel('Year')
ax1.set_ylabel('Concentration (ppm)')
ax1.set_xticks(np.arange(1960, 2020, 10))
ax1.set_yticks(np.arange(310, 400, 10))

ax1.plot(time[0:186], co2[0:186], label='SIO', color="blue")
ax1.plot(time[185:], co2[185:], label='CDIAC', color="red")

ax1.legend(loc='upper left')

plt.savefig('ex38.png')
```



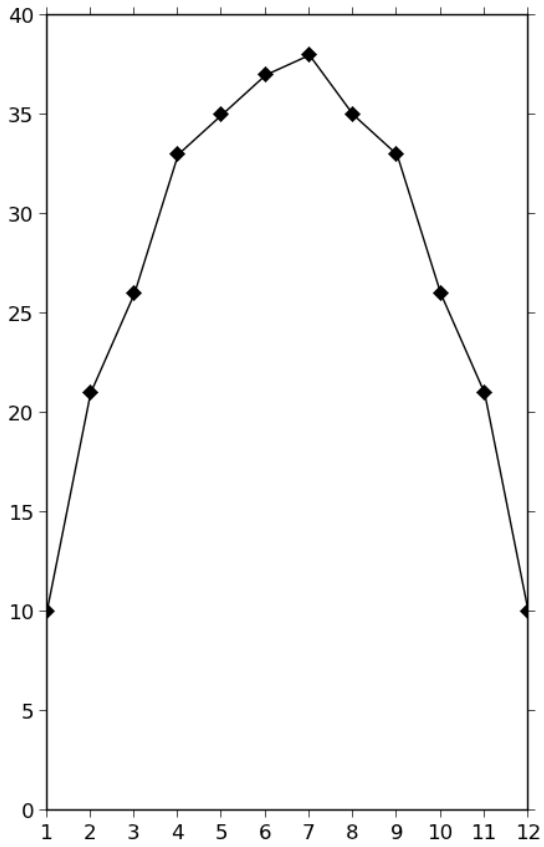
```
#!/usr/bin/env python
import matplotlib.pyplot as plt
import numpy as np
from netCDF4 import Dataset as ncfile

nc = ncfile('/opt/graphics/cfplot_data/amo.nc')
time=nc.variables['year'][:]
index=nc.variables['index'][:]

fig = plt.figure(figsize=(11, 8))
ax1 = fig.add_subplot(111)
ax1.axis([1870, 2003, -0.3, 0.3])
ax1.tick_params(direction='out', which='both')
ax1.set_xlabel('Year')
ax1.set_ylabel('Index')
ax1.set_xticks(np.arange(1880, 2010, 10))
ax1.set_yticks(np.arange(-0.3, 0.4, 0.1))

ax1.plot (time, index, color='black')
ax1.fill_between(time, index, 0, where=index<0, color='blue', interpolate=True)
ax1.fill_between(time, index, 0, index>0, color='red', interpolate=True)

plt.savefig('ex40.png')
```

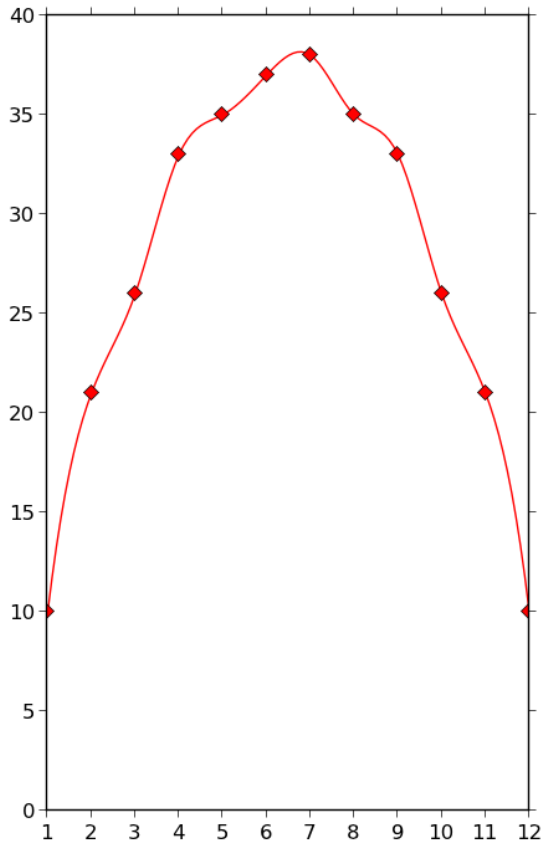


```
#!/usr/bin/env python
import matplotlib.pyplot as plt
import numpy as np
from netCDF4 import Dataset as ncfile

nc = ncfile('/opt/graphics/cfplot_data/gdata.nc')
lats=nc.variables['lat'][:]
p=nc.variables['p'][:]
temp=nc.variables['temp'][:,:,:]
temp=np.mean(temp, axis=2)
profile1=temp[:, 12]
profile2=temp[:, 36]
profile3=temp[:, 60]

fig = plt.figure(figsize=(11, 8))
ax1 = fig.add_subplot(111)

ax1.axis([-90, 90, 1000, 0.316])
ax1.tick_params(direction='out', which='both')
ax1.set_xlabel('Latitude (degrees)')
ax1.set_ylabel('Pressure (mb)')
ax1.set_yscale('log')
ax1.set_xticks(np.arange(-90, 120, 30))
ax1.set_yticks([1000,100,10,1])

ax2 = ax1.twinx()
ax2.tick_params(direction='out', which='both')
ax2.axis([-90, 90, 0, 56.78])
ax2.set_ylabel('Height (km)')
ax2.set_yticks(np.arange(0, 60, 10))
ax1.plot(profile1, p, label='60S', color="red")
ax1.plot(profile2, p, label='Equator', color="green")
ax1.plot(profile3, p, label='60N', color="blue")
ax1.legend(loc='upper left')

plt.savefig('ex41.png')
```

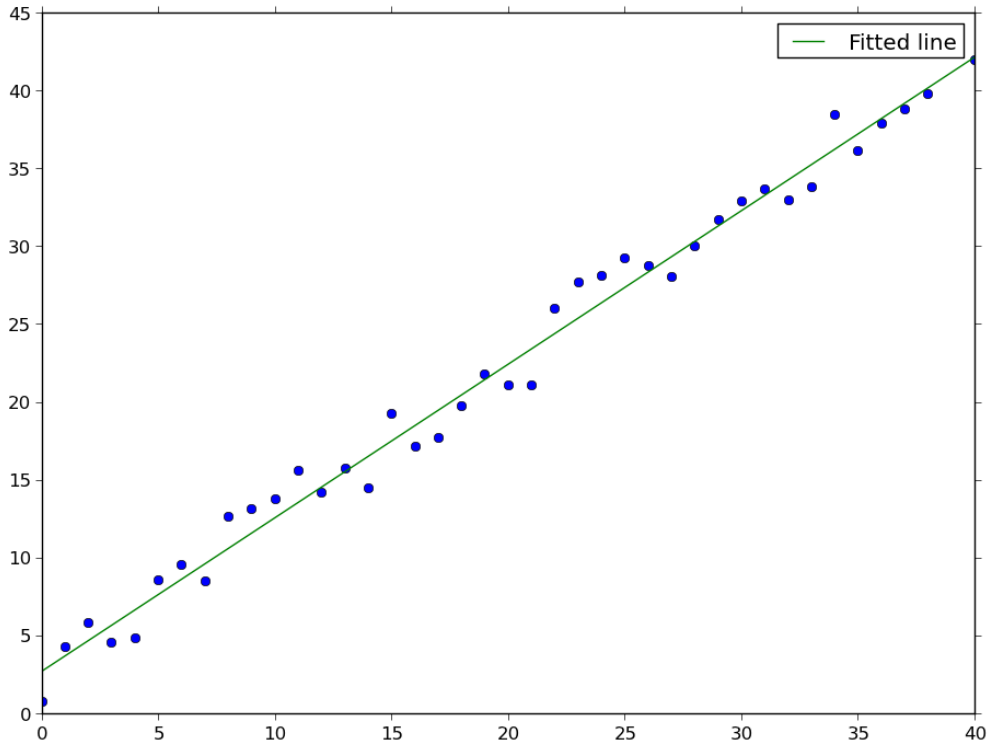

60S
Equator
60N
Pressure (mb)
Height (km)
Latitude (degrees)
10^0
10^1
10^2
10^3
−90
−60
−30
0
30
60
90
0
10
20
30
40
50

```
#!/usr/bin/env python
import matplotlib.pyplot as plt
import numpy as np
from scipy import interpolate

fig = plt.figure(figsize=(11, 8))
ax1 = fig.add_subplot(121)
ax1.axis([1, 12, 0, 40])
ax1.tick_params(direction='out', which='both')
ax1.set_xticks(np.arange(1,13,1))
ax1.set_yticks(np.arange(0, 45, 5))

xpts=np.arange(1,13,1)
ypts=[10, 21, 26, 33, 35, 37, 38, 35, 33, 26, 21, 10]
ax1.plot(xpts, ypts, color='black', marker='D')

ax2 = fig.add_subplot(122)
ax2.axis([1, 12, 0, 40])
ax2.tick_params(direction='out', which='both')
ax2.set_xticks(np.arange(1,13,1))
ax2.set_yticks(np.arange(0, 45, 5))

#tck = interpolate.splrep(xpts,ypts,s=0)
newx=np.arange(111.0)/10+1
#newy= interpolate.splev(newx,tck,der=0)
f=interpolate.interp1d(xpts, ypts, kind='cubic')

newy=f(newx)

ax2.plot(newx, newy, color='red')
ax2.plot(xpts, ypts, 'D', color='red')

plt.savefig('ex43.png')
```

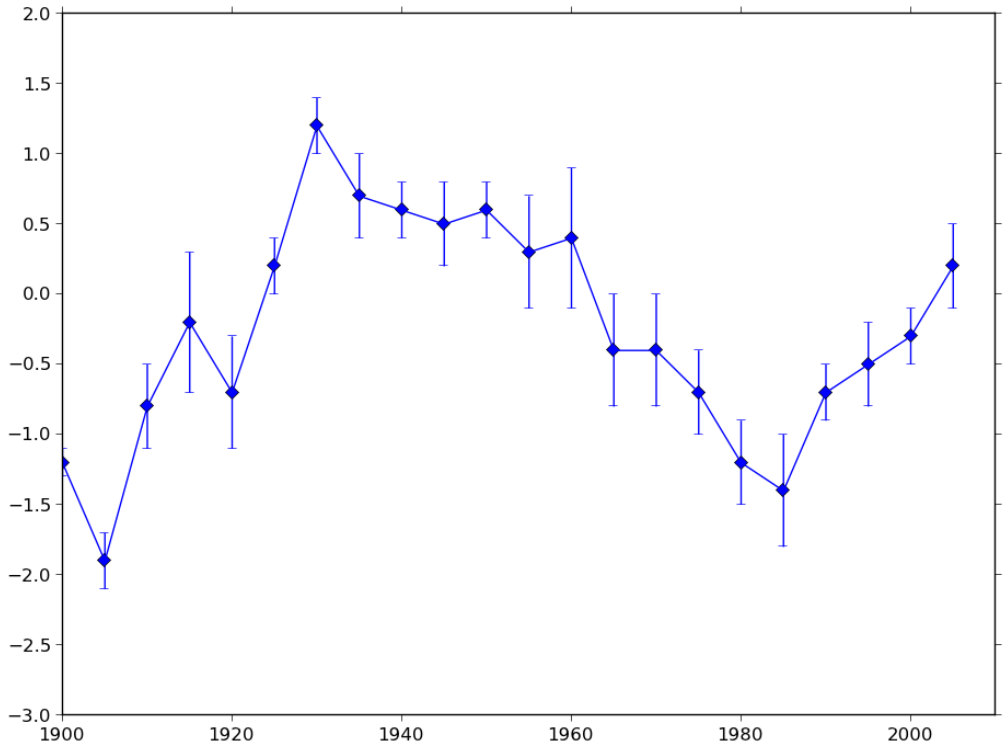

40
35
30
25
20
15
10
5
0
1 2 3 4 5 6 7 8 9 10 11 12


40
35
30
25
20
15
10
5
0
1 2 3 4 5 6 7 8 9 10 11 12

```
#!/usr/bin/env python
import matplotlib.pyplot as plt
import numpy as np

fig = plt.figure(figsize=(11, 8))
ax1 = fig.add_subplot(111)

ax1.axis([0, 40, 0, 45])
ax1.tick_params(direction='out', which='both')
ax1.set_xticks(np.arange(0, 45, 5))
ax1.set_yticks(np.arange(0, 50, 5))

xpts=np.arange(0,41,1)
ypts=np.random.rand(41)*5+xpts

#line fitting
a = np.vstack([xpts, np.ones(len(xpts))]).T
m,c=np.linalg.lstsq(a, ypts)[0]

ax1.plot(xpts, ypts, 'o')
ax1.plot(xpts, m*xpts+c, label='Fitted line')
ax1.legend()

plt.savefig('ex44.png')
```



```
#!/usr/bin/env python
import matplotlib.pyplot as plt
import numpy as np
from netCDF4 import Dataset as ncfile

#read in data
nc = ncfile('/opt/graphics/cfplot_data/errorplot.nc')
years=nc.variables['years'][:]
temp_anom=nc.variables['temp_anom'][:]
errors=nc.variables['errors'][:]

fig = plt.figure(figsize=(11, 8))

ax1 = fig.add_subplot(111)
ax1.axis([1900, 2010, -3, 2])
ax1.tick_params(direction='out', which='both')
ax1.set_xticks(np.arange(1900, 2020, 20))
ax1.set_yticks(np.arange(-3, 2.5, 0.5))

ax1.plot(years, temp_anom, 'D', color='blue')
ax1.errorbar(years, temp_anom, yerr=errors, color='blue')

plt.savefig('ex45.png')
```



```
#!/usr/bin/env python
import matplotlib.pyplot as plt
import numpy as np
from netCDF4 import Dataset as ncfile
nc = ncfile('/opt/graphics/cfplot_data/rainhist.nc')
years=nc.variables['years'][:]
rainfall=nc.variables['rainfall'][:]

fig = plt.figure(figsize=(11, 8))

ax1 = fig.add_subplot(111)
ax1.axis([1970, 2008, -600, 300])
ax1.tick_params(direction='out', which='both')
ax1.set_xticks(np.arange(1970,2010,5))
ax1.set_yticks(np.arange(-600, 400, 100))
ax1.set_xlabel('Year')
ax1.set_ylabel('Rainfall Surplus/Deficit (mm)')

for rain in np.arange(-600, 400, 100):
   ax1.plot([1970, 2010], [rain, rain], color='black', zorder=-1)

ax1.bar(years, rainfall, color='grey')

plt.savefig('ex46.png')
```